June 1990 Thesis/~~Dissertation~~

Efficiency of the Residue Number System for Computing Discrete Fourier Transforms

Thomas P. Dennedy

AFIT Student at: Massachusettes Institute of Technology

AFIT/CI/CIA -90-116

AFIT/CI
Wright-Patterson AFB OH 45433

UNCLASSIFIED

116

CSDL-T-1046

EFFICIENCY OF THE RESIDUE NUMBER SYSTEM FOR COMPUTING DISCRETE FOURIER TRANSFORMS

by

Thomas P. Dennedy

June 1990

Master of Science Thesis
Massachusetts Institute of Technology

The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, Massachusetts 02139

Efficiency of the Residue Number System for Computing Discrete Fourier Transforms

by

Thomas P. Dennedy
B.S.E.E., United States Air Force Academy
(1988)

Submitted in Partial Fulfillment
of the Requirements for the
Degree of
Master of Science
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
June, 1990

Signature of Author _____
Department of Electrical Engineering and Computer Science, May 11, 1990

Certified by _____
Associate Professor Bruce R. Musicus
Thesis Supervisor, Department of Electrical Engineering and Computer Science

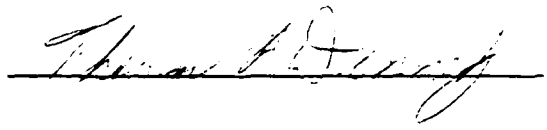
Certified by _____
Paul R. Blasche
Company Supervisor, CSDL

Accepted by _____
Professor Arthur C. Smith, Chair, Department Committee on Graduate Students

This report was prepared at the Charles Stark Draper Laboratory, Inc. under Contract 90-0-C29.

Publication of this report does not constitute approval by the Draper Laboratory or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.



Permission is hereby granted by The Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

Efficiency of the Residue Number System for Computing Discrete Fourier Transforms

by

Thomas P. Dennedy

Submitted to the
Department of Electrical Engineering and Computer Science
on May 11, 1990 in partial fulfillment of the requirements
for the Degree of Master of Science.

Abstract

The residue number system, or RNS, is analyzed in detail and compared against a conventional two's complement system for the problem of computing Discrete Fourier Transforms (DFTs) via the Winograd Fourier Transform Algorithm (WFTA.) The analysis shows that in a side-by-side comparison, the size and speed advantages of RNS cannot compensate for the high overhead required for conversion and scaling. The residue number system, or RNS, is a system of representing integers by their remainders, or residues, after division by a predetermined set of relatively prime integers. Operations such as addition, subtraction, and multiplication can be performed with modular arithmetic on these residues in independent channels, such that it is a carry-free system. RNS can exploit efficient ROM layouts by building arithmetic units out of ROMs. The main disadvantage of RNS is that scaling and conversion back to RNS require a lengthy series of operations. The WFTA is found to be the best algorithm for doing DFTs in RNS because it reduces the need for scaling by nesting the necessary multiplications into one layer, such that there is only one layer of coefficients that increase the range of the output data.

An area-time metric for a custom VLSI layout is used to compare RNS adders and multipliers to conventional binary components. It is shown that in a strict side-by-side comparison, with no rounding allowed, RNS can outperform two's complement significantly. However, when RNS is compared aagainst a two's complement system not constrained by the RNS requirement to avoid scaling, the two's complement can use rounding arithmetic to beat RNS. Since the WFTA was found to be the most ideal algorithm for doing DFTs in the RNS, it is concluded that RNS does not provide a real advantage for doing DFTs. A new system is proposed, in which DFTs are performed using distributed arithmetic with ROM lookup tables.

Thesis Supervisor: Professor Bruce R. Musicus

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I am very grateful for all the people in my life who have helped me since I came to MIT two years ago. When I first came, I was not sure how the couple of years I had ahead of me would pass, and where I would be at the end. But the end is here, and the companionship and support I have received have made it enjoyable. I would first and foremost like to thank my advisor Bruce Musicus, for giving me the time and attention needed to make this thesis possible. Many thanks also to Paul Blasche, for his guidance and friendship.

I am very grateful to Draper Laboratory for its support, financial and otherwise. Many thanks to Joan Morrison and John Sweeney in the education office. I am especially thankful for the effort Dr. David Burke put into making this program possible. Thanks to the many other people here who have given me help and assistance— especially the secretaries Shelly, Pam, Leonore, Margaret, and Sandra. Thanks also to the others in the office for their aid and advice: Ira, Peter, Scot, Frank, Steve, and Seth. I feel like I'm finally figuring things out, but leaving before I can really make use of it and give something back.

Many thanks to the people with whom I have shared office space, for your friendship and companionship: Jerry, Jeff, John, Kimo, Rachel, Vince, and especially my good friend Kelly. Your company and sincere thoughts have been the source of many interesting discussions, debates, and delays in finishing this thesis. I will miss you all. Thanks for the many other friendships I have made these past two years; there are too many names to mention here.

Thanks to my roommates, John, Todd, and Bill, for your friendship, care, and making our "domicile" a great place to live and learn. Your patience, tolerance, and

trust are hard to match.

I cannot fail to mention my friends and instructors at the U.S. Air Force Academy, for their support and motivation in helping me get to MIT. Without you, I would not have had this opportunity to continue my education. Special thanks to Ms. Fern Kinion in the Graduate Awards office for the help she gave me and continues to give others pursuing advanced degrees.

Thanks to the friends I have in many far away places, for their friendship and hospitality when I needed to get away. Right about now, I'm especially thankful for the ones I'll see in L.A. this weekend— my old friend Dave and my new friend Kari— for a promising vacation that's motivated me to finish this by May 11.

Last, but certainly not least, is my family. I have been seeing less of you over the past few years, but you are always in my mind. Your actions and thoughts were suppportive and motivational when I was down, and appreciate when I was doing well. Thank you Mom, Dad, Ann, Bill, Jim, Sandy, Dan, Karen, Peggy, and Bobby.

Contents

1 Introduction **11**

1.1 The RNS in Digital Signal Processing 12

1.2 Computing DFTs with RNS Arithmetic 17

2 Using the RNS in Digital Signal Processing **19**

2.1 Conversion From Two's Complement to RNS 19

2.2 RNS Processing 20

2.3 Conversion From RNS to Two's Complement 23

3 Performance of RNS Hardware **26**

3.1 Performance Measures 26

3.1.1 Speed 27

3.1.2 Size 27

3.1.3 Area-Time Metric 28

3.1.4 ROM Layout vs. Full Adder 28

3.2 RNS Computational Elements 45

3.2.1 RNS Adders 45

3.2.2 RNS Multipliers 51

3.3 Conventional Computational Elements 57

3.3.1 Adders 57

3.3.2 Multipliers 57

3.4 Side-By-Side Comparison 58

3.4.1 Adders 59

3.4.2 Multipliers 59

3.4.3 Registers 60

4 DFT Algorithms **62**

4.1 Cooley-Tukey FFT 63

4.2 Prime Factor Algorithm 65

4.3 Winograd Fourier Transform Algorithm 68

4.4 Comparison of Algorithms 70

5 Performance Comparison **73**
5.1 System Area-Time Products 73
5.2 Analysis of Problems 74

6 Conclusions **82**
6.1 Performance of RNS 82
6.2 New Ideas 83
6.3 Suggestions for Further Research 86

A Matlab M-files **88**

Bibliography **112**

List of Figures

3.1 Construction of ROM. 29
3.2 NOR array. 31
3.3 Column tree decoder. 32
3.4 Layout of NOR array. 33
3.5 Layout of column decoder. 35
3.6 Size of ROM (λ^2). 40
3.7 Speed of ROM (ns). 41
3.8 Transmission gate adder. 42
3.9 Area of ROM relative to a full adder. 45
3.10 Speed of ROM relative to a full adder. 46
3.11 Area-time of ROM relative to a full adder. 47
3.12 Modulo m_i adder using ROM look-up table. 47
3.13 Modulo m_i adder using correction ROM. 48
3.14 Modulo m_i adder using two binary adders. 49
3.15 Area-time products for three modular adders. 50
3.16 Modulo m_i multiplier using index calculus. 52
3.17 Modulo m_i multiplier using quarter squares identity. 53
3.18 Modified modulo m_i multiplier using quarter squares identity. 54
3.19 Modulo m_i multiplier using modular adders. 55
3.20 Area-time products for four modular multipliers. 56
3.21 Integer multiplier. 57
3.22 RNS/Integer multiplier area-time ratio for different numbers of moduli. 60

4.1 Flowgraph of an 8-point FFT. 64
4.2 Two-factor prime factor algorithm. 67
4.3 A 15-point WFTA. 69

5.1 Area-time products for systems performing WFTAs. 76
5.2 Comparison of two's complement and RNS systems using modified moduli selection. 77
5.3 Comparison of two's complement and RNS systems using dynamic range penalty in moduli selection. 79
5.4 Comparison of two's complement and RNS systems using smaller dynamic range two's complement. 80

6.1 General block diagram of system using distributed arithmetic. 83
6.2 Radix-4 butterfly using distributed arithmetic. 84

List of Tables

4.1 Number of real operations required for length-N DFT of real data (double for complex data)[3]. 67
4.2 Comparison of operation counts for DFTs of complex data. 71

Chapter 1

Introduction

Computing the Discrete Fourier Transform (DFT) of a data sequence is a common operation in digital signal processing. However, the time required to perform transforms can be a significant bottleneck in applications where there is a large number of data points, such as in image processing and radar signal processing. There exist many algorithms for computing DFTs, the most common being the Cooley-Tukey Fast Fourier Transform (FFT) algorithm. Since then, however, several other algorithms have been developed to reduce the number of computations required, but with a less regular structure than the FFT. The most prominent of these are the Good-Winograd Fourier Transform Algorithm (GWFTA, also known as the Prime Factor Algorithm) and the Winograd Fourier Transform Algorithm (WFTA), which reduce the number of multiplies required in the transform [3].

There has likewise been a great deal of research done on specialized high-speed architectures for performing these computations [18, 1, 20, 9, 2]. Specifically, there has been some interest in doing these calculations in the residue number system (RNS), in which integers are represented by their value modulo a predetermined set of small integers [5, 6, 19]. This number system has been shown to have properties that result in significant speed advantages in digital signal processing applications, where most of the computations required are multiplications and additions.

This thesis will analyze and compare RNS and two's complement systems in terms of size and speed for a custom layout to determine if RNS is inherently better for solving the DFT problem. Previous work has emphasized the speed advantage of RNS over conventional binary arithmetic, ignoring the increased speed possible in conventional binary arithmetic through the use of pipelined or parallel VLSI components. Therefore, an area-time product is a better metric for comparison and will be used in this thesis. This chapter will introduce the residue number system, and explain why it may be better for performing DFTs and what its disadvantages are. Chapter 2 explains in more detail the requirements for doing arithmetic in the RNS and the parts of such a system. Chapter 3 discusses the choice of using an area-time product to evaluate computational systems, and it compares two's complement components to RNS components in order to determine the relative advantage of RNS at the adder and multiplier level. Chapter 4 explains the three most efficient algorithms for computing DFTs and describes how the WFTA is the most ideal algorithm for RNS. Chapter 5 uses the area-time metric to compare two's complement and RNS systems designed for doing different size WFTAs. The conclusions, summed up in Chapter 6, are that RNS does not provide a clear advantage, in terms of the area-time metric, over two's complement for the DFT problem. Other approaches to the problem and applications of the RNS are suggested for further research.

1.1 The RNS in Digital Signal Processing

The residue number system, or RNS, is an integer coding system which has been shown to have potential speed advantages in systems designed for digital signal processing. A residue system represents an integer by a number of remainders, or residues, after division of the integer by a set of given integers. This is also called modular arithmetic because the remainder is the representation of the integer in a modulo n system, where n is the number by which the integer is divided. For exam-

ple, 23 could be represented by 2, 3, and 2, which result after division by 3, 5, and 7, respectively. RNS arithmetic is not new; in fact, the previous example is taken from *Suan-ching*, written by Sun Tzu in the third century [16]. His work describes a method for converting the residues back to the integer. Today, this procedure is appropriately termed the Chinese Remainder Theorem.

The potential advantage of RNS-based digital systems is that once integers are represented by their remainders modulo a given set of integers, operations such as addition, subtraction, and multiplication can be performed independently and in parallel on the different remainders. This arithmetic is done modulo n, where n is the integer upon which the corresponding residues are based. Since a large integer can be represented by several smaller residues, and the arithmetic operations on these residues can be performed independently and in parallel, the RNS has potential speed advantages because the residue digits are smaller yet they require no carries between them. The speed advantages are most promising in digital signal processing where the RNS-compatible operations of addition, subtraction, and multiplication are most common.

A residue number system is defined by a set of integers $\{m_1, m_2, \ldots, m_L\}$ which are pairwise relatively prime. (The greatest common factor of any two is 1.) Any integer X in the range $[0, M-1]$, where $M = m_1 \cdot m_2 \cdot \ldots \cdot m_L$, can be uniquely represented by the set of residues $\{x_1, x_2, \ldots, x_L\}$, where

$$x_i = X \bmod m_i = |X|_{m_i}, i = 1, \ldots, L. \tag{1.1}$$

Soderstrand et al. explain that for a signed number system, the legitimate range $[0, M-1]$ is divided into positive and negative regions. For M odd, the permitted range is $[-(M-1)/2, (M-1)/2]$ with negative integers mapped to $[(M+1)/2, M-1]$ of the legitimate range above. Likewise, for M even, the permitted range is $[-M/2, (M/2-1)]$ with negatives mapped to $[M/2, M-1]$ of the legitimate range [12].

Arithmetic operations in RNS are very simple for addition, subtraction, and mul-

tiplication. If X, Y, and Z are represented in the RNS by

$$\begin{aligned} X &\Rightarrow \{x_1, \ldots, x_L\} \\ Y &\Rightarrow \{y_1, \ldots, y_L\} \\ Z &\Rightarrow \{z_1, \ldots, z_L\} \end{aligned}$$

then the operation $X * Y = Z$ maps to

$$(x_i * y_i) \bmod m_i = z_i, i = 1, ..., L,$$

where $*$ represents addition, subtraction, or multiplication. General division is not possible because the set of integers is not closed over division. Scaling a number in the RNS will be discussed later. The residue digit results alone are not very useful until they are converted back to an integer. There are two well known techniques for converting the residue digits back to an integer. They both involve quite a few operations and will also be discussed later.

Techniques for implementing RNS-based systems have been studied in great detail [12]. There are generally three stages: converting data to its RNS equivalent, performing the necessary computations to process the data, and converting the results back to radix-2 binary numbers. The first operation, converting to a residue representation, is discussed by Jenkins in his description of an RNS digital filter. The conversion is done most easily with a ROM lookup table for each modulus [7]. The size of the table is within reason because the number of possible entries is limited to the size of the modulus.

Many systems have been designed for implementing the two basic RNS operations of addition and multiplication.[12, Part III] One of the simplest ways of doing addition is again to use a memory lookup table, where the address is formed by the concatenation of the two residues. The memory will have 2^{2n} words with n bits per word, where n is the number of bits in each residue. Soderstrand has explained how the size of the memory can be reduced if the addition is done by first adding the

residue digits with a binary adder and then using a memory with an $n+1$ bit address to correct the result for the given modulus. Other systems eliminate the need for the extra memory hardware by using normal binary adders and a correction factor to get the RNS result [11]. This is done most easily if moduli are of the form $2^n, 2^n+1$, and 2^n-1 because of their proximity to a power of 2.

Multiplication in RNS can be just as simple as addition. If lookup tables are used for addition, they can also be used for multiplication, so that the necessary hardware and speed will be the same as for addition. Soderstrand [13] and Jullien [8] present two other approaches, which decrease the size of the memory lookup tables in exchange for some additions. A fourth way of doing multiplication is a more conventional bitwise multiplication. The multiplicand is multiplied successively by each bit of the multiplier and the result is doubled and added to the result from the next least significant bit. This method requires modular adders and, therefore, is more simple if the moduli are of the form $2^n, 2^n+1$, and 2^n-1.

Two well-known methods exist for converting residues back to integers. Szabo and Tanaka describe them in their comprehensive text on RNS [14]. The first is the Chinese Remainder Theorem:

$$X = \sum_{i=1}^{L} \left(\hat{m}_i \left| \frac{x_i}{\hat{m}_i} \right|_{m_i} \right) \bmod M, \text{ where } \hat{m}_i = M/m_i. \tag{1.2}$$

Although this is the shorter of the two, it requires the use of a modulo-M adder, where M is the dynamic range. The other method is to do a mixed-radix conversion on the RNS digits. An integer is represented in the mixed-radix system by

$$X = a_N \prod_{i=1}^{L-1} m_i + \ldots + a_3 m_1 m_2 + a_2 m_1 + a_1, \tag{1.3}$$

where the a_i's are called the mixed-radix coefficients. These coefficients can be found through a sequence of $L-1$ subtractions and $L-1$ multiplications on the original residues. The integer X is found by adding the terms in the above expression. Any necessary division or scaling in the RNS can also be performed by converting to the mixed-radix form since it is a weighted number representation.

Scaling is a problem which must be dealt with in RNS systems because of the potential problem of exceeding the dynamic range of a system. The problem is that in RNS, the magnitude of an integer depends on all its residues together, so each residue digit cannot be scaled independently. Scaling is different from general division because it implies division by a given constant and rounding. Szabo and Tanaka describe the procedure most commonly used, which is to scale by one of the moduli [14]. In order to ensure that the rounded division will result in an integer, the remainder for the modulus by which the integer is being scaled is subtracted from the RNS representation. The RNS digits are then multiplied by the multiplicative inverse of the scaling modulus and a base extension algorithm, similar to mixed-radix conversion, is used to determine the new residue for the scaling modulus. This algorithm requires significantly fewer computations if the moduli are of the popular $2^n, 2^n + 1$, and $2^n - 1$ form.

The speed advantages of RNS arithmetic extend from the fact that operations on the residues can be carried out independently. Since the residue digits are limited by the size of the moduli and there is no carry information sent from one residue digit to another, residue systems should perform faster than conventional radix-2 systems due to the reduction of carry propagations. The drawbacks are that

1. general division is not possible since division is not closed over the set of integers,

2. scaling, a more specific type of division, is very inefficient, and

3. some time must be invested in converting integers to and from the residue system.

Hence, the speed advantage of RNS is most promising when the number of simple arithmetic operations required by the application algorithm is large with respect to any required scaling or conversion operations.

1.2 Computing DFTs with RNS Arithmetic

Using RNS arithmetic to compute DFTs raises issues that are not so great a concern with a radix-2 weighted number system. The greatest of these issues is the problem of scaling, because RNS is a system for encoding *integers*. Any DFT algorithm requires the multiplication of the input data by twiddle factors of the form $e^{-j\frac{2\pi}{N}nk}$, whose real and imaginary parts are a cosine and a sine. These values are always less than 1, so that encoding them in RNS requires scaling them up by the number of bits of desired resolution. With the twiddle factors all represented by integers greater than 1, the dynamic range grows with the transform size. A residue system would therefore require a dynamic range larger than that of the original data to allow for integer growth before scaling. This is not a problem in a radix-2 system because any desired scaling and truncation can be done by shifting the decimal point and throwing away the least significant digits to make room for the digits with a greater weight. As explained in the previous section, however, scaling RNS values requires a significant number of computations.

The size of this scaling problem depends on the actual DFT algorithm and moduli set being used. Taylor has shown that the WFTA would be most suitable for RNS because it reduces the number of multiplies to a minimum and, hence, reduces the number of scaling operations [17]. Whether or not the RNS can improve the area-time product of DFT calculations is a question that will be addressed in this thesis.

There are two main problems with determining whether or not an RNS-based design can outperform a similar system using conventional arithmetic. The first is that there is no standard, optimal design for a system that computes DFTs. Most of these specialized hardware designs are highly pipelined, but other than that, they use many different architectures and algorithms depending on the application and the desired performance. This leads to the second problem, in that there is no typical application with a set of requirements that is representative of all systems requiring the use of DFTs. DFTs vary in the size of the transform and in the desired accuracy

and speed of the computation. Accordingly, a system can be designed to optimize its performance for a given application, even though it may be quite inefficient for another.

For these reasons, the first goal of this thesis is to show that the area-time metric represents the most significant constraints on systems used to compute DFTs. These constraints include speed, size (in terms of silicon area), and power as a function of the transform length and the desired accuracy. The performance of the best algorithm and hardware implementation in a standard technology base will be evaluated using this metric to determine performance characteristics of specific DFTs implemented with an RNS architecture.

Chapter 2

Using the RNS in Digital Signal Processing

This chapter will expand on the requirements for doing digital signal processing in the residue number system. As explained in Chapter 1, there are three parts to an RNS system:

1. conversion of data from two's complement or other binary format to RNS,
2. processing the data in the RNS, and
3. conversion of the data from RNS to original binary format.

These three parts of the problem will be addressed before examining how the RNS compares to conventional two's complement arithmetic.

2.1 Conversion From Two's Complement to RNS

The first part of the processing in RNS is to calculate the residue representation of each piece of data for the set of moduli being used. Since the ultimate goal of using the RNS is to increase the speed of a system, this conversion must be done as quickly

as possible. Jenkins discusses a common and efficient solution, which is to use high-speed ROMs to look up the residues for each modulus independently. For relatively small moduli, each of these ROMs would be of a manageable size[7]. If we let b be the number of bits in the original integer X, and we wish to determine $x_i = |X|_{m_i}$, where m_i has b_i bits, then the ROM must be of size $2^b \times b_i$ bits. Although b_i can be kept relatively small (3–7 bits), the original dynamic range determined by b may be large, requiring a very large ROM with many small words. An ideal solution to this problem is to divide the problem into smaller pieces. The b-bit number may be divided into several smaller numbers, such as the least significant and most significant $\frac{b}{2}$ bits. These smaller numbers can be looked up in ROMs independently, weighted by proper powers of two, and then added together in a modulo m_i adder. Techniques for modular addition will be discussed in the next chapter.

2.2 RNS Processing

The processing of data in the RNS parallels the processing of data for the same problem in two's complement notation, for the RNS-compatible operations of addition, subtraction, and multiplication. Hardware implementation of these operations will be discussed in the following chapter.

Scaling is a process that must be treated much differently in the RNS than in two's complement. Whereas two's complement data may be scaled by shifting bits, one place value for each factor of two, scaling a number in the RNS requires a lengthy series of operations. Scaling is different from general division in that data is divided by a predetermined constant and then rounded. Szabo and Tanaka explain the most efficient type of scaling, in which the data is scaled by one or more moduli. The first step in the scaling process is to round the number to a multiple of the intended scaling factor. Let X be the integer to be scaled, and let Y be the scaling factor. If we round

X down to a multiple of Y, then the scaling problem is reduced to computing

$$\frac{X - |X|_Y}{Y}. \tag{2.1}$$

Since Y will be one of the moduli or the product of several, the existence of its multiplicative inverse modulo the other moduli is guaranteed by the requirement that the moduli be relatively prime. The residues of X are therefore multiplied by $|\frac{1}{Y}|_{m_i}$ within each of the RNS channels. The final step in the scaling problem is to perform a base extension to determine the residues of the scaled result for the moduli by which X was scaled. The base extension is similar to a mixed-radix conversion on the moduli by which X was not scaled; the mixed-radix conversion will be discussed in the next section [14].

The RNS scaling process will be demonstrated by an example, taken from Szabo and Tanaka, in which a positive integer is scaled by two moduli. Let the moduli be $m_1 = 2, m_2 = 3$, $m_3 = 5$, and $m_4 = 7$. The integer

$$X = 89 \leftrightarrow \{1, 2, 4, 5\}$$

will be scaled by $15 = 3 \cdot 5$, yielding the result $z = \frac{X}{15}$. The solution is shown below.

Moduli:	2	3	5	7		
Residue representation of X	1	2	4	5		
Subtract $\|X\|_3 = 2$	0	2	2	2		
	1	0	2	3	$\overset{2,3,5,7}{\leftrightarrow}$	$X - \|X\|_3$
Multiply by $\left\|\frac{1}{3}\right\|_{m_i}$	1	-	2	5		
	1	-	4	1	$\overset{2,5,7}{\leftrightarrow}$	$\frac{X-\|X\|_3}{3}$
Subtract $\|X\|_5 = 4$	0	-	4	4		
	1	-	0	4	$\overset{2,5,7}{\leftrightarrow}$	$\frac{X-\|X\|_3}{3} - \|X\|_5$
Multiply by $\left\|\frac{1}{5}\right\|_{m_i}$	1	-	-	3		
Enter 0 in missing columns for base extension	1	0	0	5	$\overset{2,7}{\leftrightarrow}$	$\frac{1}{5}\left(\frac{X-\|X\|_3}{3} - \|X\|_5\right) = \left\lfloor\frac{X}{15}\right\rfloor$
Subtract 1	1	1	1	1		
	0	2	4	4		
Multiply by $\left\|\frac{1}{2}\right\|_{m_i}$	-	2	3	4		
	-	1	2	2		
Subtract 2	-	2	2	2		
	-	2	0	0		

Then $\left|\frac{1}{2}|z|_3 + 2\right|_3 = 0$ and $\left|\frac{1}{2}|z|_5 + 0\right|_5 = 0$; hence, $|z|_3 = 2$ and $|z|_5 = 0$.
Therefore, the residue representation of the scaled result $\left\lfloor\frac{89}{15}\right\rfloor$ is $\{1, 2, 0, 5\} \leftrightarrow 5$.
This example shows that it takes L multiplies and L additions (in each of the RNS channels) to scale a number in the RNS, not including the final subtraction and multiplication required to determine the final two residues of z. Remember that L is the number of moduli in the system[14].

There are several observations that should be made concerning this scaling algorithm. The first is that the number of operations is independent of the number of moduli by which the integer is being scaled, except for the final subtraction and multiplication to solve for z, which may be done in parallel anyway. All of the opera-

tions are also performed within the separate RNS modular arithmetic channels. The process is complicated slightly if we are interested in rounding to the nearest integer instead of simply rounding down and if we need to scale negative integers as well as positive ones. These cases are explained in more detail by Szabo and Tanaka [14].

2.3 Conversion From RNS to Two's Complement

There are two methods for converting an integer from the RNS to a conventional two's complement format. These techniques are called the Chinese Remainder Theorem and the mixed-radix conversion and were presented in Chapter 1. Examples of these conversion methods will be given to help explain the processes.

The Chinese Remainder Theorem (CRT) is the classical conversion algorithm. The equation from Chapter 1 is repeated here:

$$X = \sum_{i=1}^{L} \left(\hat{m}_i \left| \frac{x_i}{\hat{m}_i} \right|_{m_i} \right) \bmod M, \text{where } \hat{m}_i = M/m_i. \tag{2.2}$$

The following example of the CRT is taken from a tutorial by Taylor [16]. Let $m_1 = 3$, $m_2 = 4$, and $m_3 = 5$, such that $M = 60$. The problem will be to convert X={1,0,4} back to an integer. The values of $\hat{m}_i$ and their multiplicative inverses can be precalculated:

$$\begin{aligned} \hat{m}_1 &= 20, \quad \hat{m}_1^{-1} = 2 \\ \hat{m}_2 &= 15, \quad \hat{m}_2^{-1} = 3 \\ \hat{m}_3 &= 12, \quad \hat{m}_3^{-1} = 3. \end{aligned}$$

The solution is

$$\begin{aligned} X &= |(20 \cdot |1 \cdot 2|_3) + (15 \cdot |0 \cdot 3|_4) + (12 \cdot |4 \cdot 3|_5)|_{60} \\ &= 4. \end{aligned}$$

The CRT requires one layer of modular multiplications and then a sequence of L multiplications and $L-1$ adds modulo M. The requirement for a modulo-M adder

is the main disadvantage of the CRT. An RNS system will have components to do modular arithmetic for the factors of M, but not for M itself.

The equation for the mixed-radix conversion (MRC) is

$$X = a_N \prod_{i=1}^{L-1} m_i + \cdots + a_3 m_1 m_2 + a_2 m_1 + a_1. \tag{2.3}$$

The coefficients of the form a_i are called the mixed-radix coefficients, and each one may take on values of $0, \ldots, m_i - 1$. The mixed-radix representation of a number is a weighted number system because the value of the integer is a weighted sum of the coefficients. This representation can therefore be used for magnitude comparison and sign detection when negative integers are encoded. Szabo and Tanaka explain how these coefficients are found through a series of nested subtractions and divisions. If Equation 2.3 is first evaluated modulo m_1, it is clear that

$$|X|_{m_1} = a_1,$$

so that a_1 is simply the first residue digit. The next coefficient is found by noting that

$$\left| \frac{X - a_1}{m_1} \right|_{m_2} = a_2.$$

The division by m_1 is possible because its multiplicative inverse exists for all the other moduli, by virtue of the fact that the moduli are selected to be relatively prime. The remaining mixed-radix coefficients can be found by repeating this subtraction and division, as demonstrated in the following example from Szabo and Tanaka.

Moduli:	8	5	7	3	
Residue representation of X	$3 = a_1$	4	2	1	$a_1 = \|X\|_8 = 3$
Subtract $a_1 = 3$	3	3	3	0	
	0	1	6	1	$\overset{8,5,7,3}{\leftrightarrow} X - a_1$
Multiply by $\left\|\frac{1}{8}\right\|_{m_i}$		2	1	2	
		$2 = a_2$	6	2	$a_2 = \left\|\left[\frac{X}{m_1}\right]\right\|_{m_2} = \left\|\left[\frac{X}{8}\right]\right\|_5 = 2$
Subtract $a_2 = 2$		2	2	2	
		0	4	0	$\overset{5,7,3}{\leftrightarrow} \frac{X-a_1}{8} - a_2$
Multiply by $\left\|\frac{1}{5}\right\|_{m_i}$		3	2		
			$5 = a_3$	0	$a_3 = \left\|\left[\frac{X}{m_1 m_2}\right]\right\|_{m_3} = \left\|\left[\frac{X}{40}\right]\right\|_7 = 5$
Subtract $a_3 = 5$			5	2	
			0	1	$\overset{7,3}{\leftrightarrow} \frac{\frac{X-a_1}{8}-a_2}{5} - a_3$
Multiply by $\left\|\frac{1}{7}\right\|_{m_i}$				1	
				$1 = a_4$	$a_4 = \left\|\left[\frac{X}{m_1 m_2 m_3}\right]\right\|_{m_4} = \left\|\left[\frac{X}{280}\right]\right\|_3 = 1$

The mixed-radix representation of X is now {1,5,2,3}. From Equation 2.3 the integer is

$$X = 1(8 \cdot 5 \cdot 7) + 5(8 \cdot 5) + 2(8) + 3(1) = 499.$$

The mixed-radix conversion process requires $L - 1$ layers of subtractions and $L - 1$ layers of multiplies within the RNS components. This is a count of the layers of operations because the individual RNS channels will perform operations in parallel. Conventional arithmetic is then required to multiply each coefficient by its weight and add the components ($L - 1$ multiplies and the same number of adds.) While the operation count is higher than for the CRT, most of the arithmetic is done within the RNS channels and no modulo-M adder is required.

Chapter 3

Performance of RNS Hardware

In this chapter, a set of performance measures for analysis of architectures for computing DFTs will be established. These measures will be applied to the basic hardware components used in computing DFTs to determine the performance advantage of RNS hardware at the building block level. These performance measures will then be used in the next chapter as the basis for a comparison between binary weighted number and RNS architectures for computing entire DFTs.

3.1 Performance Measures

Some primary concerns of a system designer for signal processing are speed, size, cost, and power. The last three are directly related to each other for a given technology base, so cost and power can be reasonably predicted given some measure of size. Speed is the other main concern and must defined such that it reflects only an architecture's ability to solve a problem quickly and not parameters such as the clocking scheme or operation definition. Since the goal is to compare design architectures, we can choose either high-level measures (gate delay and gate count) or low-level measures (such as silicon area and actual time) for a specific technology, under the assumption that these measures would scale consistently for a different technology.

3.1.1 Speed

The primary goal in measuring speed is to compare the impact of choosing the RNS over conventional arithmetic. In this analysis, the same algorithm, architecture, and building blocks will be used to design two systems for computing DFTs, except that RNS components will be substituted for two's complement components in one system. Speed will be measured by the time to compute one DFT, which shall be called latency. Pipelining and other techniques used to increase throughput will not be used because their impact would be similar for both systems. The situation in which it would not be similar is when pipeline registers are placed inside the arithmetic operators instead of between them. While allowing data to pass through at a higher rate, pipelining does not reduce the actual time spent performing the necessary operations on the data. In fact, the latency would increase due to the additional time of passing data through the pipeline registers.

3.1.2 Size

The size of a system is a parameter which generally increases with its speed. For a given technology and design approach, a system's speed can be increased by using more hardware, but it is desirable to improve upon a system's speed without a proportional increase in size. Other parameters of a system that increase with size are cost and power; although the relationship may not be linear, these two parameters can be reasonably estimated from size. Thus, the metrics used for comparing systems will be limited to speed and size.

Determining how to measure size is a somewhat more difficult problem. Whatever measure is used, it should be one that allows a relative comparison of two designs independent of the common technology used to implement them. One such measure is gate count. Given the hardware components required for a design (adders, registers, *etc.*), a library of standard cells can be used to determine gate count. One drawback to this approach is that not all hardware translates directly into gates, an example

being ROMs. ROMs are frequently used to implement arithmetic in RNS devices, so the relative size between a ROM and other logic elements will have to be established. The approach chosen here will be to develop an expression for the size and speed of a ROM in terms of the size and speed of a full adder for a 3μm CMOS technology. A full adder is used for comparison because it is the basic computational element for addition and multiplication. After expressing these measures relative to those of a full adder, conversions may be made to units such as equivalent gate count and delay.

3.1.3 Area-Time Metric

Comparing the performance of two or more system designs is complicated by the fact that they generally have different sizes and different times. Sometimes one design dominates the others in the sense that it has a smaller size and is faster; more generally, however, the designs cannot be directly ranked by both measures simultaneously. To resolve this difficulty, we propose using an area-time product to objectively compare designs in terms of a single metric. Combining area and time into this single metric assumes that there is a tradeoff in area and time for any component such that their product remains approximately constant. The backing for this claim is the idea that twice the hardware can be used to do twice as many operations in any given time interval, thus halving the time per operation. This metric does not address the cost of interconnecting these components, but it does reflect inherent space and time requirements for computations performed by them.

3.1.4 ROM Layout vs. Full Adder

This section will develop a set of equations to approximate the size and speed of a ROM relative to a full adder for a 3 μm CMOS technology.

Layout of a ROM

The construction of a typical ROM is shown in Figure 3.1. The ROM will be analyzed

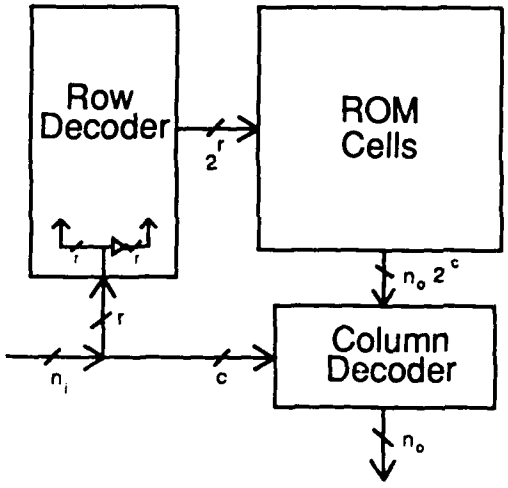


Figure 3.1: Construction of ROM.

in terms of the three main parts in the figure to determine an expression for the silicon area and time delay. These measures will depend on the geometry of the ROM; ROMs are usually laid out in the shape of a square, as opposed to a rectangle with different dimensions. Let n_i be the number of bits in the input, or address, and n_o be the number of bits in each output word. For a ROM of $2^{n_i} \times n_o$ bits the ideal length of each side of the memory portion would be $\sqrt{n_o 2^{n_i}}$ bits. Each column in the ROM contains an n_o-bit word, so the number of columns from which the address bits must select is

$$\frac{\sqrt{n_o 2^{n_i}}}{n_o}. \tag{3.1}$$

The number of address bits in the column selector is given by

$$c = \left[\frac{1}{2} n_i - \frac{1}{2} \log_2 n_o\right], \tag{3.2}$$

so that the number of address bit in the row decoder is given by

$$r = \left[\frac{1}{2} n_i + \frac{1}{2} \log_2 n_o\right], \tag{3.3}$$

where $[x]$ represents the nearest integer to x (rounded). Care must be taken in rounding so that $c + r = n_i$.

The analysis of the three parts of the ROM is discussed by Hodges and Jackson. Both the row decoder and the core of the ROM are usually designed as a NOR array, as shown in Figure 3.2. The pull-up devices may be either pMOS precharge or nMOS depletion-mode devices [4]. For the row decoder, there are two rows of transistors for each of the r row address bits. The row decoder has 2^r output lines that connect to the rows of the ROM core.

The NOR arrays get their name because they operate just like multiple-input NOR gates. The output of each column in the array is the result of NORing all the inputs to the nMOS devices. These devices are connected in parallel to ground, so that a high input to any of them results in the column being discharged. Either the nMOS devices or the contacts to them are placed selectively to generate the desired output for each possible set of inputs. For the row decoder, each column will be a word select line for the ROM. The transistors in that column are connected to the proper set of address input bit lines, or their inverses, so that only one column remains high for each possible address. The row decoder is rotated 90 degrees so that the columns run horizontally; this allows the column lines to be connected directly to the row lines of the ROM. Only one row line in the ROM will be high for any set of inputs, and transistors are placed appropriately in each row of the ROM to generate the correct output word.

The simplest and most common type of column decoder is the tree decoder, shown in Figure 3.3. This is simply a tree of nMOS transistors used as pass gates to select from one of 2^c columns, where c is the number of column address bits. There must be one of these trees for each bit in the ROM's output word.

The area of a ROM is determined by the dimensions of the diagram in Figure 3.1. The width of the entire ROM is approximately equal to the width of the row decoder plus the width of the main memory portion. The height is approximately equal to the height of the main memory plus the height of the column decoder. The inverters for both the row and column address bits should fit within the space in the lower left

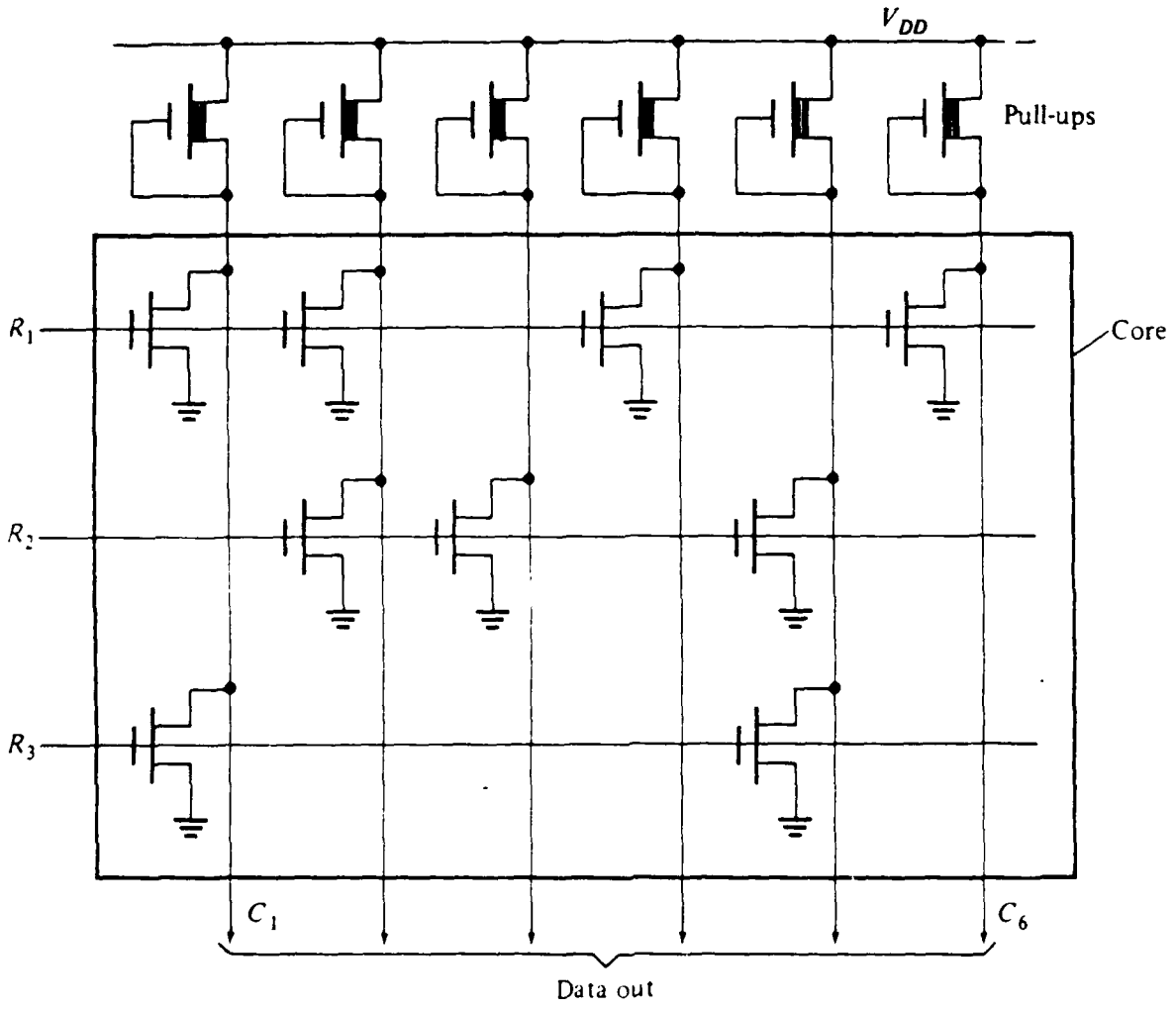


Figure 3.2: NOR array.

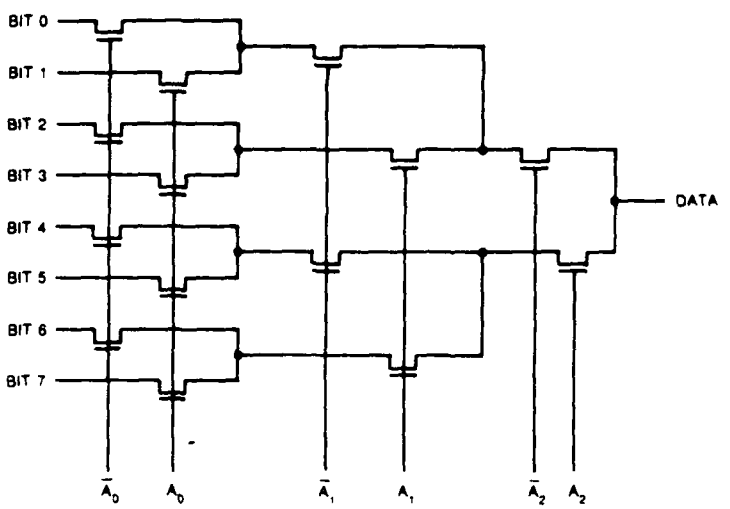


Figure 3.3: Column tree decoder.

corner.

The sizes of the row decoder and memory cells can be found by studying the layout of a NOR array, shown in Figure 3.4. This figure shows the most compact layout of the array on a lambda scale for a set of MOSIS scalable design rules. We will temporarily use R to represent the number of rows and C the number of columns in the NOR array. There must be 9λ between each column, as shown, plus approximately 2λ on the left side for spacing from the row decoder or other devices. The resulting expression for the width of the NOR array is

$$W_{NOR} = (9C + 2)\lambda. \tag{3.4}$$

The height of the array depends on the number of rows, R. Figure 3.4 shows that there is 16λ for each pair of adjacent rows. Additionally, there is 19λ on top for precharge or loading devices and 8λ on bottom for separation from the column decoder. The final expression for the height of the NOR array is

$$\begin{aligned} H_{NOR} &= (16R/2 + 8 + 19)\lambda \\ &= (8R + 27)\lambda. \end{aligned} \tag{3.5}$$

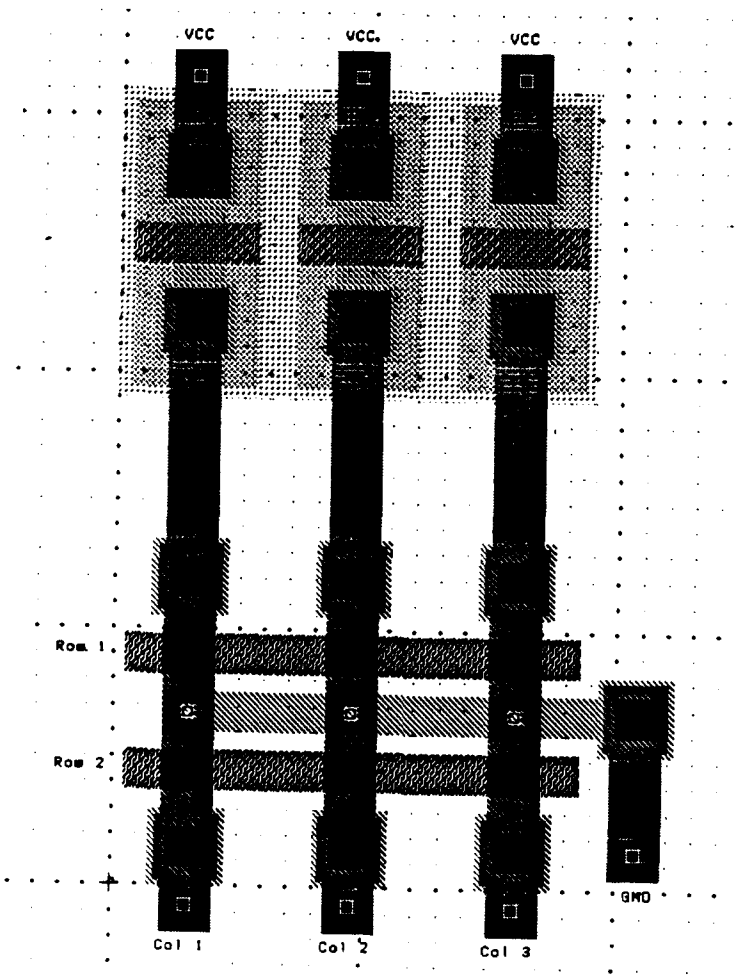


Figure 3.4: Layout of NOR array.

An expression for the height of the column decoder is now needed so that the height of the whole ROM can be determined. The exact width of the column decoder is not important because it is less than the width of the ROM core, as shown in Figure 3.1. Figure 3.5 shows the layout of the column decoder. The height of the column decoder can be expressed as

$$H_{cd} = 18c\lambda, \tag{3.6}$$

where, again, c is the number of address bits in the column decoder.

The size of the entire ROM can now be found. The width is equal to the height of the row decoder (before it is rotated) plus the width of the ROM core:

$$\begin{aligned} W_{ROM} &= (8(2r) + 27)\lambda + (9(2^c n_o) + 2)\lambda \\ &= (16r + 9(2^c)n_o + 29)\lambda. \end{aligned} \tag{3.7}$$

The height of the ROM is equal to the height of the ROM core plus the height of the column decoder:

$$\begin{aligned} H_{ROM} &= (8(2^r) + 27)\lambda + 18c\lambda \\ &= (2^{(r+3)} + 18c + 27)\lambda \end{aligned} \tag{3.8}$$

The total area can now be written as

$$\begin{aligned} A_{ROM} &= W_{ROM} \times H_{ROM} \\ &= (16r + 9(2^c)n_o + 29)(2^{(r+3)} + 18c + 27)\lambda^2. \end{aligned} \tag{3.9}$$

The timing analysis for the ROM begins with the following assumptions for the process parameters:

Symbol	Parameter	Value
λ	$\frac{1}{2}$ length of min transistor	1.5 μm
R_{poly}	resistance of polysilicon	50Ω/square
C_{ox}	capacitance of thin oxide	$500\frac{\mu F}{m^2}$
C_d	source or drain capacitance	$C_g/3$

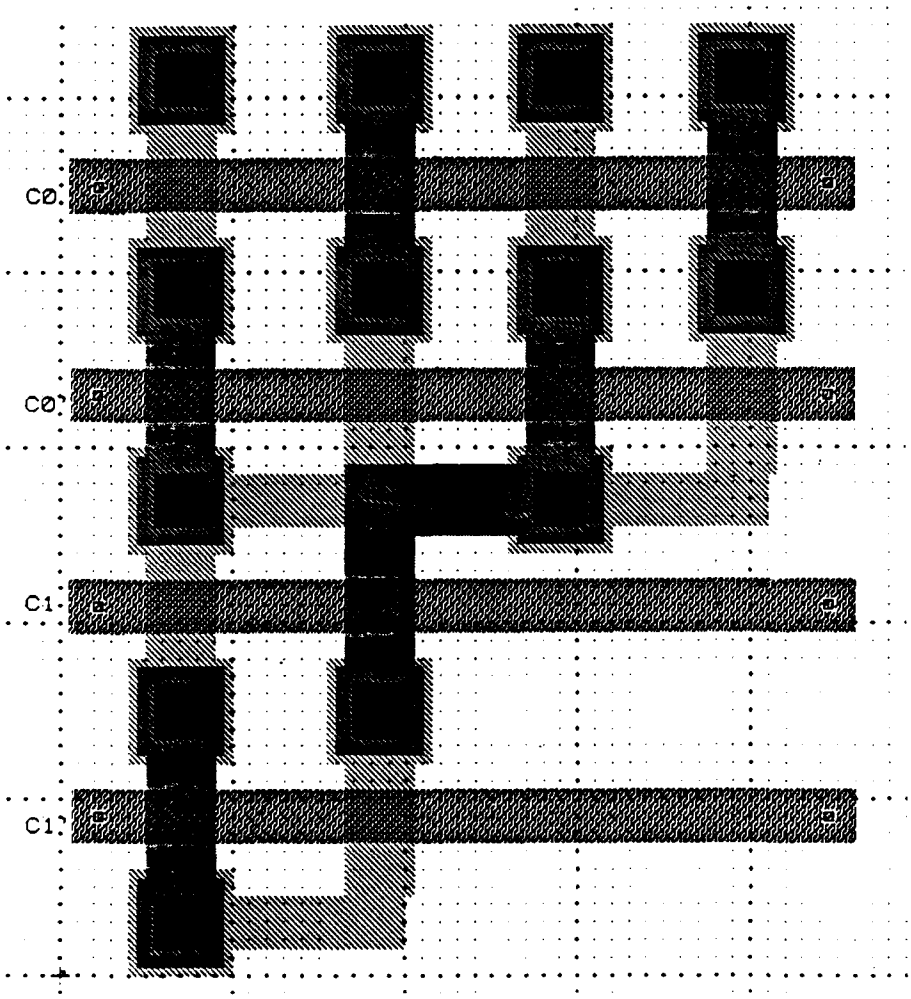


Figure 3.5: Layout of column decoder.

The gate capacitance for a minimum size transistor, of $4.5\mu m \times 3.0\mu m$ is

$$\begin{aligned} C_g &= 4.5\mu m \times 3.0\mu m \times 500\mu F/m^2 \\ &= 6.75fF. \end{aligned} \tag{3.10}$$

The parameters K_n and K_p, which are used in the equations that model MOS devices, are estimated below for a $4.5\mu m \times 3.0\mu m$ transistor:

$$\begin{aligned} K_n &= \mu_n C_{ox} \frac{\text{width}}{\text{length}} \\ &= 26\frac{\mu A}{v^2} \end{aligned} \tag{3.11}$$

$$\begin{aligned} K_p &\approx K_n/2 \\ &= 13\frac{\mu A}{v^2}. \end{aligned} \tag{3.12}$$

The access time for the ROM can be expressed as

$$T_{access} = T_{row} + T_{rom} + T_{col}, \tag{3.13}$$

where

$$\begin{aligned} T_{row} &= \text{delay through row decoder} \\ T_{rom} &= \text{delay through ROM cells} \\ T_{col} &= \text{delay through column decoder.} \end{aligned}$$

The calculations for T_{row} and T_{rom} are similar because both are delays through NOR arrays. Hodges and Jackson analyze the delay through a NOR array by expressing it as the sum of the switching time for the row lines and the charging/discharging of the column lines. Let t_r be the delay through a row in the NOR array and let t_c be the delay through a column. The delay through a row can be approximated by the following equation for the 50% output transition for a uniformly distributed RC line with a step input:

$$t_r = .38RC, \tag{3.14}$$

where R is the total resistance and C is the total capacitance of the distributed line. The delay through a column in the NOR array can be approximated by the time for the p-type device at the top of a column to charge the column capacitance to the 50% point [4]:

$$t_c = \frac{C\Delta V}{I_{avg}}. \tag{3.15}$$

The values of R and C are calculated by using the parameters above and the geometry of the layouts in Figures 3.4 and 3.5.

The delay through the row decoder, T_{row}, will now be calculated by determining the delays through the rows and columns within its NOR array. There are r bits in the row decoder, and each bit plus its inverse go into separate rows in the NOR array, yielding $2r$ rows. There are 2^r columns, each driving a separate word select line in the ROM core. For an array of $2r$ rows by 2^r columns, the delay through a row is

$$\begin{aligned} t_r &= .38(2^r \times \frac{9}{2} \times 50\Omega)(2^r \times 6.75fF) \\ &= 5.77 \times 10^{-4} \times 4^r \text{ns}. \end{aligned} \tag{3.16}$$

To calculate the column delay, the column capacitance and average charging current for a column must be found. Each column will have a transistor connecting it to each of the input bits or its inverse, for a complete row decoder. Thus, each column has r of the $2r$ rows connected to it, resulting in a column capacitance of

$$r \times \frac{6.75fF}{3}. \tag{3.17}$$

The worst case delay is when the depletion mode device at the top of a column must charge the column from 0 to 5 volts. The average current, I_{avg}, is the average of the current at 0 volts and the current at 2.5 volts output. The parameter K for the depletion device is assumed to be $\frac{1}{4}$ that for a normal nMOS device, and the threshold voltage is assumed to be $V_T = -3v$. At 0 volts, the depletion device has 5 volts across it, but $V_G - V_T = 5v - (-3v) = 8v$, so it is in its linear region. Thus, the current at the beginning of the charging is

$$I_{0V} = \frac{K_n}{4}((5v - (-3v))5v - \frac{5v^2}{2})$$

$$= 179\mu A. \tag{3.18}$$

At an output level of 2.5 volts, the transistor is still in the linear region, so the current at this point is

$$\begin{aligned} I_{2.5v} &= \frac{K_n}{4}((5v-(-3v))2.5v - \frac{2.5v^2}{2}) \\ &= 110\mu A. \end{aligned} \tag{3.19}$$

The delay through the columns in the row decoder is now

$$\begin{aligned} t_c &= \frac{r \times \frac{6.75fF}{3} \times 2.5v}{\frac{179\mu A+110\mu A}{2}} \\ &= 0.0389 \times r \text{ ns.} \end{aligned} \tag{3.20}$$

The total delay through the row decoder is now

$$T_{row} = (5.77 \times 10^{-4} \times 4^r + 0.0389r)\text{ns.} \tag{3.21}$$

The delay through the ROM memory cells can be calculated in a similar manner. The size of the array is now 2^r rows by $n_o 2^c$ columns. The delay through a row is given by

$$\begin{aligned} t_r &= .38(n_o 2^c(\frac{9}{2})50\Omega)(n_o 2^c 6.75fF) \\ &= 5.77 \times 10^{-4} n_o^2 4^c \text{ns.} \end{aligned} \tag{3.22}$$

The delay through a column is given by

$$\begin{aligned} t_c &= \frac{2^r \times \frac{6.75fF}{3} \times 2.5v}{\frac{179\mu A+110\mu A}{2}} \\ &= 0.0389 \times 2^r \text{ns.} \end{aligned} \tag{3.23}$$

The total delay through the ROM cells is

$$T_{rom} = 5.77 \times 10^{-4} n_o^2 4^c + 0.0389 \times 2^r \text{ns.} \tag{3.24}$$

The delay through the column decoder can be approximated by using Equation 3.15 and assuming that the source and drain capacitances must be discharged by

an nMOS transistor whose length is c times the length of each individual transistor. The total capacitance is

$$C = (1 + 2c)\frac{6.75fF}{3}. \tag{3.25}$$

Since the length to width ratio of the effective transistor is c times normal, the currents before and halfway through discharging are

$$\begin{aligned} I_{5V} &= \frac{1}{2c}K_n(5v - .6v)^2 \quad \text{(saturation)} \\ &= \frac{251.7}{c}\mu\text{A} \end{aligned} \tag{3.26}$$

$$\begin{aligned} I_{2.5V} &= \frac{1}{c}K_n((5v - .6v)2.5v - \frac{(2.5v)^2}{2}) \\ &= \frac{204.8}{c}\mu\text{A}. \end{aligned} \tag{3.27}$$

The average current is

$$I_{avg-n} = \frac{251.7 + 204.8}{2}\mu\text{A} = 228\mu\text{A}. \tag{3.28}$$

The delay through the column decoder is therefore

$$\begin{aligned} T_{col} &= \frac{(1 + 2c)\frac{6.75f\text{F}}{3}(2.5\text{v})(2c)}{228\mu\text{A}} \\ &= 0.0493c(1 + 2c)\text{ns}. \end{aligned} \tag{3.29}$$

The above results for area and time are summarized in Figures 3.6 and 3.7 respectively. These plots show the area and time as a function of n_o, the size of an output word. The three cases when n_i is either n_o, $n_o + 1$, or $2n_o$ will later be shown to be of primary interest for the ROMs used in RNS. The three curves, therefore, represent these three cases. The layout of a full adder is now analyzed so that a comparison can be made between it and a ROM.

Layout of a Full Adder

The schematic and layout of a typical full adder as given by Weste and Eshraghian is shown in Figure 3.8 [21]. This adder makes heavy use of transmission gates. Although

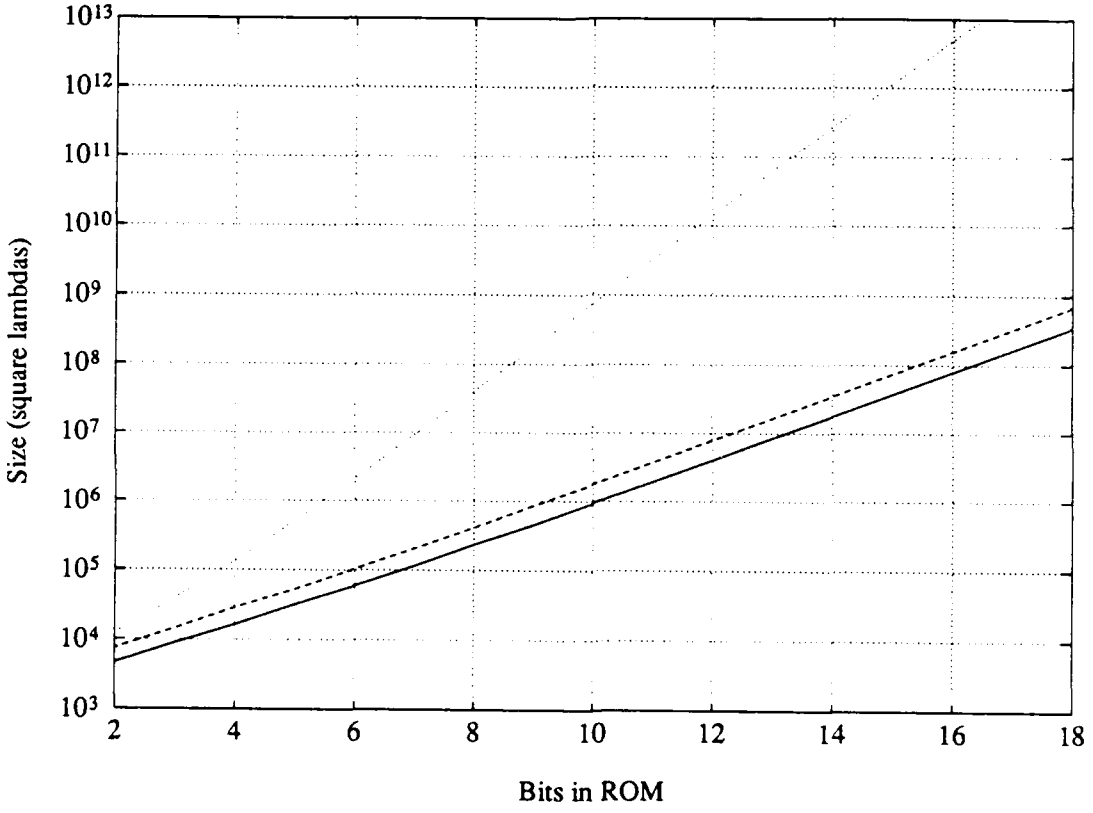


Figure 3.6: Size of ROM (λ^2).

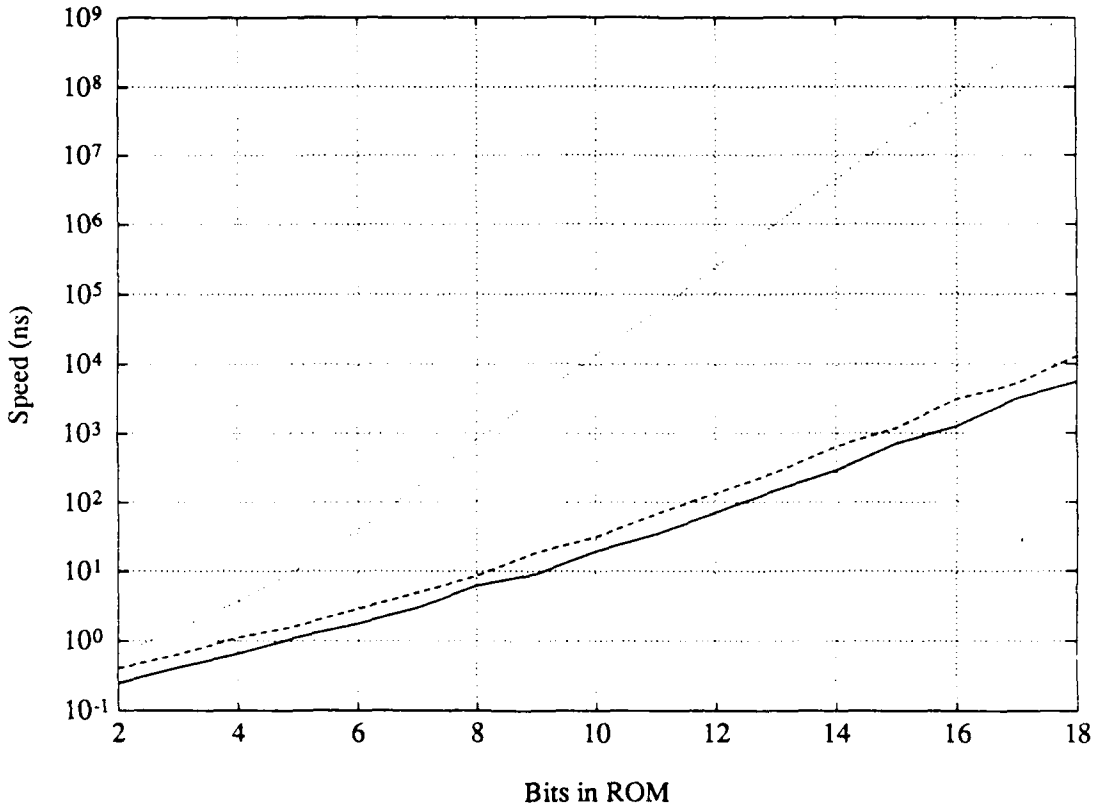


Figure 3.7: Speed of ROM (ns).

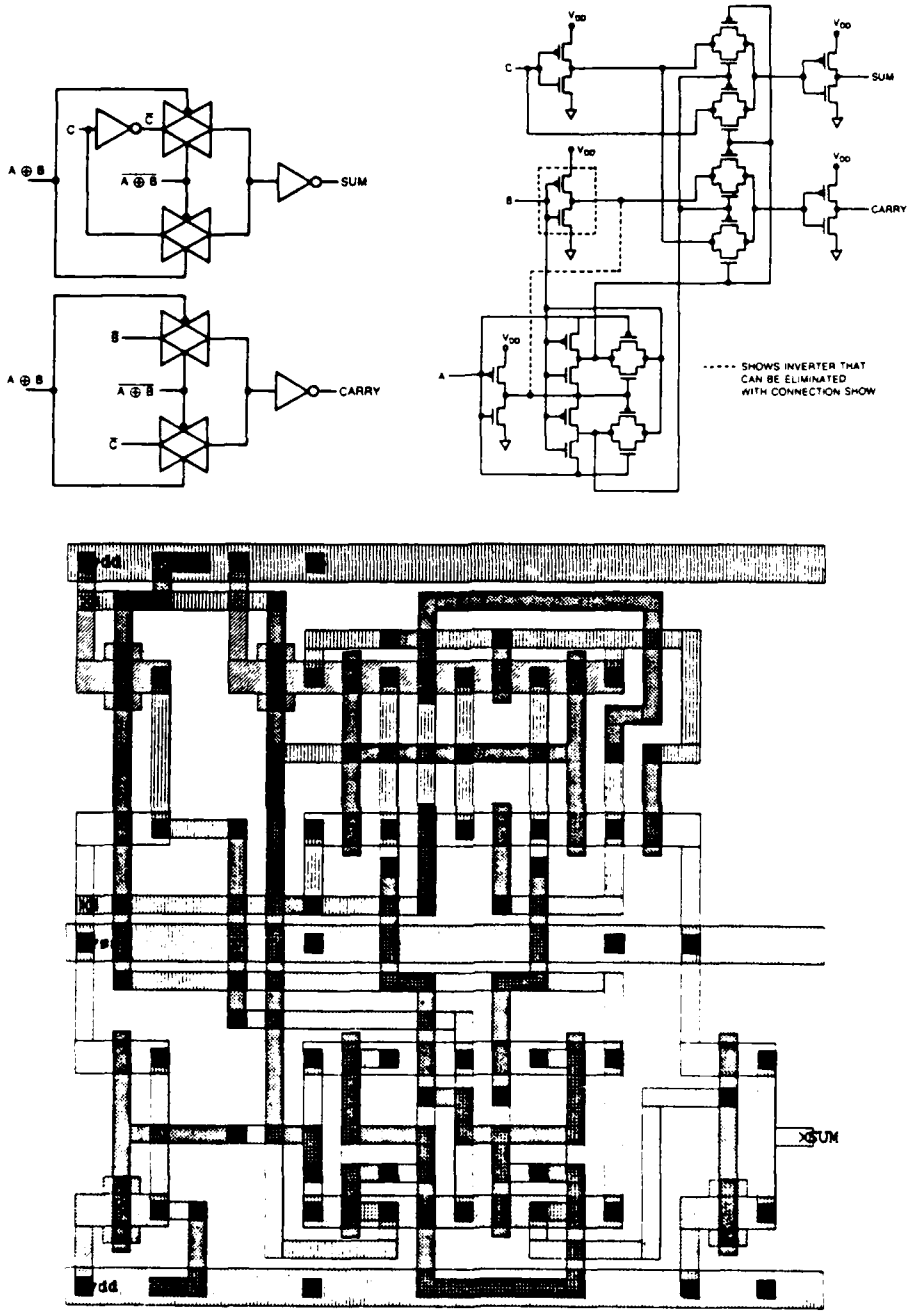


Figure 3.8: Transmission gate adder.

other full adder designs are possible, they require the same number of transistors (24) and approximately the same area. The size of this adder, using the same 3μm design rules as for the ROMs, is $76\lambda \times 101\lambda = 7,676\lambda^2$.

The delay through the full adder can be estimated using Equation 3.15. The critical delay path is from carry-in to carry-out. Figure 3.8 shows that the signal must propagate through three gates: carry-in passes through an inverter, the inverted signal passes through a pass gate controlled by the inputs A and B, and the result passes through another inverter. The worst case is when carry-in changes from 0 to 1, because then the output inverter will also change from 0 to 1. It will be shown that the second inverter has a larger capacitive load than the first, so that the worst case occurs when the second must charge the load from 0v to 5v through its p-type device.

The capacitive load on the first inverter comes from the 4 source/drains of the pass gates at that node and from the length of the polysilicon line. The capacitance of this line is

$$30\lambda \times 1.5\frac{\mu\text{m}}{\lambda} \times .2\frac{\text{pF}}{\text{mm}} = 9\text{fF}.$$

The total capacitance is now

$$C = 4\frac{6.75\text{fF}}{3} + 9\text{fF} = 18\text{fF}. \tag{3.30}$$

The average discharge current through an n-type device was previously found to be 228μA. The delay through the first inverter can now be calculated:

$$\begin{aligned} t_{inv1} &= \frac{18\text{fF} \cdot 2.5\text{v}}{228\mu\text{A}} \\ &= .197\text{ns}. \end{aligned} \tag{3.31}$$

The capacitive load on the pass gate is due to the 2 gates of the inverter it drives and the polysilicon line connecting it to the inverter. The capacitance of this line is

$$16\lambda \times 1.5\frac{\mu\text{m}}{\lambda} \times .2\frac{\text{pF}}{\text{mm}} = 4.8\text{fF}.$$

The total capacitance is

$$C = 2 \cdot 6.75\text{fF} + 4.8\text{fF} = 18.3\text{fF}. \tag{3.32}$$

A good estimate of the average current through the pass gate is the average current through an n-type device, 228μA. The delay through the pass gate is

$$\begin{aligned} t_{pass} &= \frac{18.3\text{fF} \cdot 2.5\text{v}}{228\mu\text{A}} \\ &= .201\text{ns}. \end{aligned} \tag{3.33}$$

The capacitive load on the second inverter comes from the polysilicon carry-in line of the next full adder, the 2 gates of its first inverter, plus the 2 source/drains of the pass gate also connected to the carry-in line. The capacitance of this line is

$$57\lambda \times 1.5\frac{\mu\text{m}}{\lambda} \times .2\frac{\text{pF}}{\text{mm}} = 17.1\text{fF}.$$

The total capacitance is now

$$C = 2 \cdot 6.75\text{fF} + 2\frac{6.75\text{fF}}{3} + 17.1\text{fF} = 35.1\text{fF}. \tag{3.34}$$

The average current through a p-type device is approximately one half that through an n-type, so the average charging current through this inverter is $\frac{228}{2}\mu\text{A} = 114\mu\text{A}$. The delay through the second inverter is

$$\begin{aligned} t_{inv2} &= \frac{35.1\text{fF} \cdot 2.5\text{v}}{114\mu\text{A}} \\ &= .770\text{ns}. \end{aligned} \tag{3.35}$$

The total delay from carry-in to carry-out is

$$T_{fa} = 1.168\text{ns}. \tag{3.36}$$

The area and speed of a ROM can now be normalized to a full adder by dividing the results of Figures 3.6 and 3.7 by the area and speed of a full adder as just calculated. These results are shown in Figures 3.9 and 3.10. Figure 3.11 shows the area-time product of a ROM relative to the area-time of a full adder. The efficiency of ROM layouts is apparent by noting that the area-time product of a 4-input, 2-output ROM is slightly less than the area-time of a full adder.

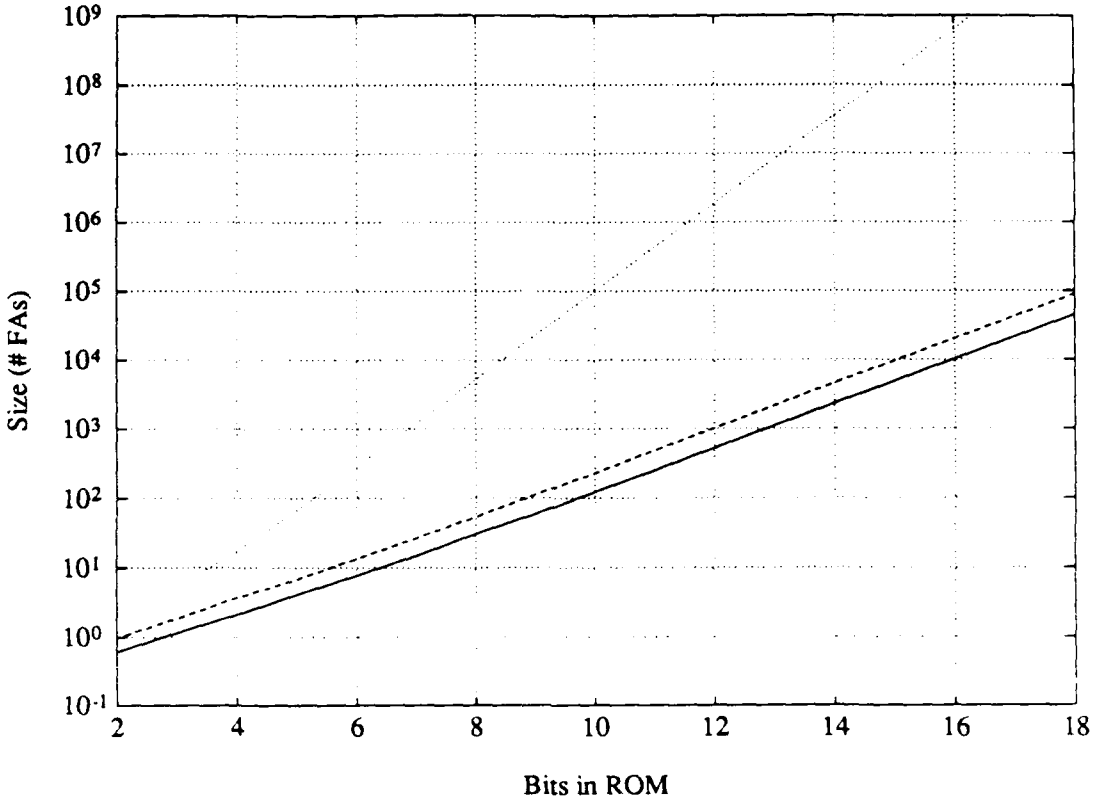


Figure 3.9: Area of ROM relative to a full adder.

3.2 RNS Computational Elements

Before delving into a performance analysis for the specific problem of performing DFTs, an analysis of the individual computational hardware elements used in this problem will be done. For DFTs, these elements are adders and multipliers.

3.2.1 RNS Adders

As mentioned in Chapter 1, there are at least three ways to implement modular addition with digital hardware. The first is to concatenate the binary representation of the residues and use the result as an address to a ROM look-up table which contains the sum, modulo the proper integer, as shown in Figure 3.12. This method can be used for multiplication also, with the result that these two operations will take the same amount of time. For an n-bit modulus, $2^{n-1} < m_i \leq 2^n$, the ROM will have 2^{2n}

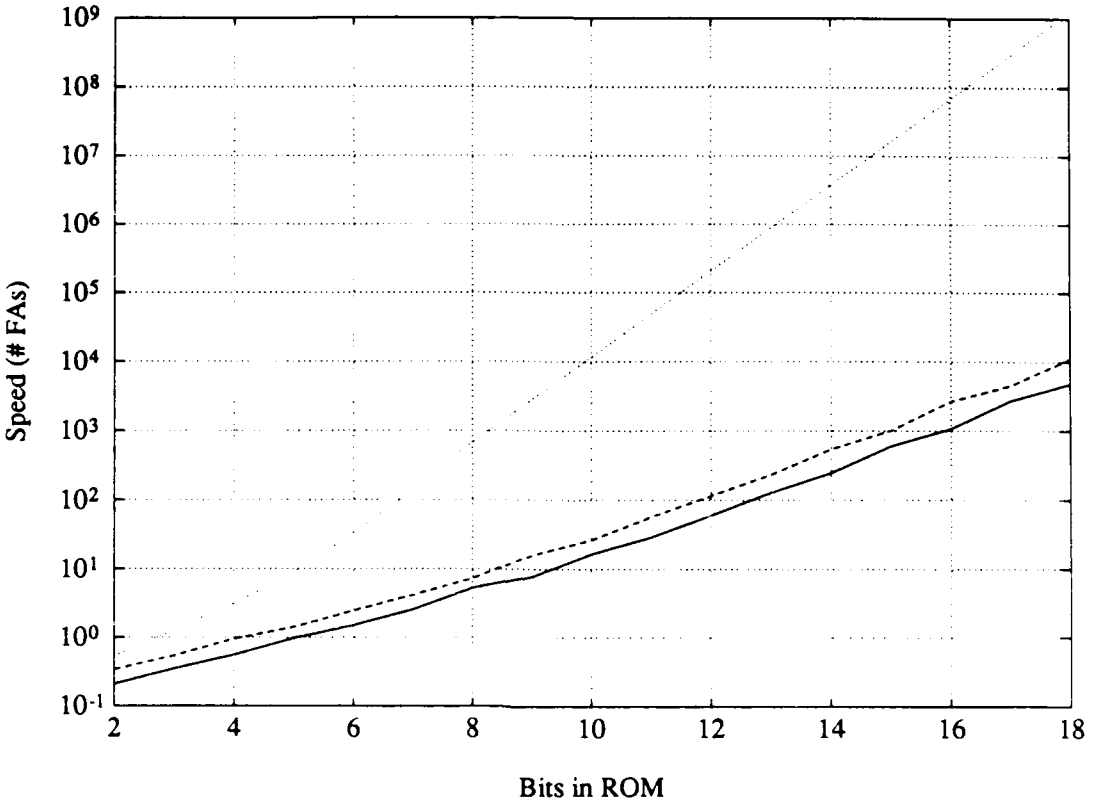


Figure 3.10: Speed of ROM relative to a full adder.

words of length n bits.

Soderstrand presents two other methods of performing modular addition [11]. The basic procedure is to add the two numbers like normal binary numbers and then to correct the result by subtracting the modulus if the result exceeds the range of numbers allowed by that modulus. There are two ways to do the correction. One method uses a ROM to correct the $n+1$-bit sum to an n-bit sum within the proper range, as shown in Figure 3.13. While still requiring a ROM, the size of the ROM has been decreased from 2^{2n} to 2^{n+1} words. Another method is to use a second n-bit adder to add the value $2^n - m_i$ to the output of the first adder. If either adder produces a carry, the corrected result is used; otherwise, the output of the first adder is used. This scheme can be derived by starting with a two's complement system that subtracts m_i if the result of the first addition is greater than or equal to m_i. Let the addends be x and y such that $0 \leq x < m_i, 0 \leq y < m_i$. In twos complement

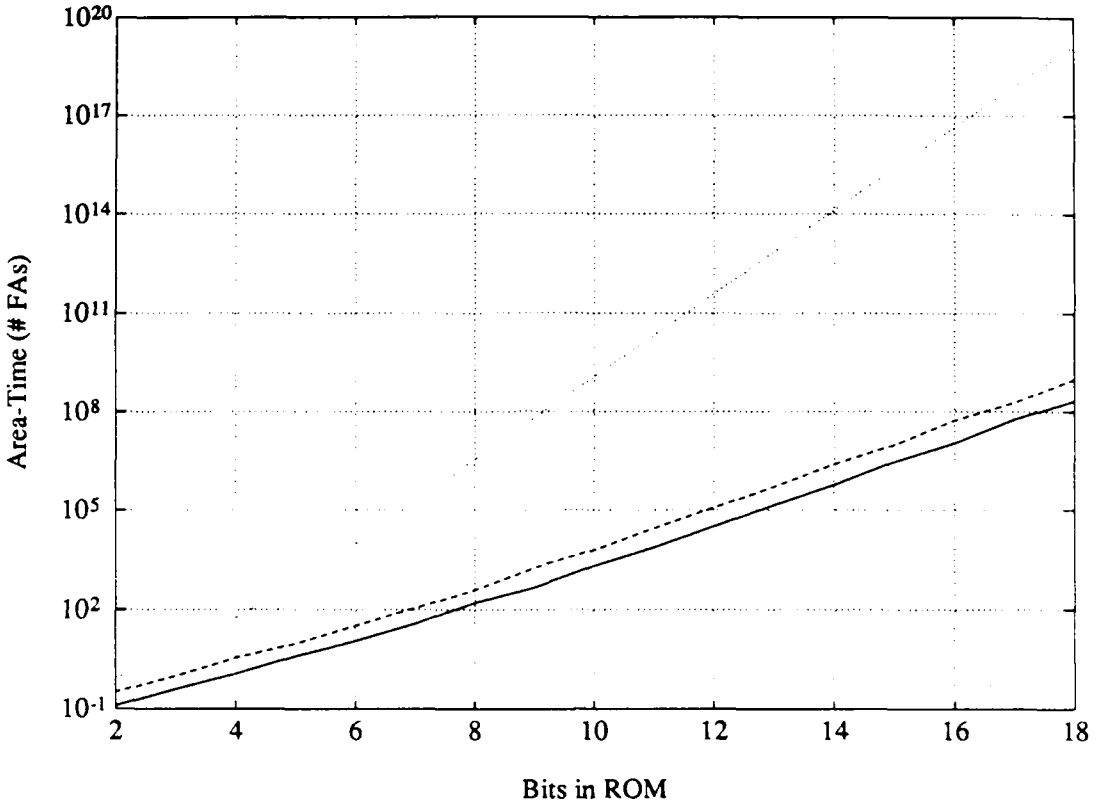


Figure 3.11: Area-time of ROM relative to a full adder.

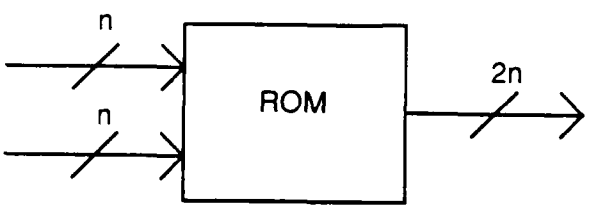


Figure 3.12: Modulo m_i adder using ROM look-up table.

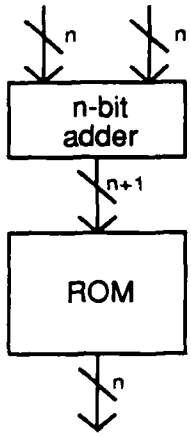


Figure 3.13: Modulo m_i adder using correction ROM.

notation these each require $n+1$ bits. The first adder computes $\dot{z} = x + y$, which requires $n+2$ bits. The next adder computes $\ddot{z} = \dot{z} - m_i$, which must fall in the range $-m_i \leq \ddot{z} \leq m_i - 1$, an $n+1$-bit twos complement number. If $\ddot{z}$ is negative, then $z = \dot{z}$; otherwise, $z = \ddot{z}$. This procedure can be simplified by noting that since negative integers are not used for x, y, and z then only n bits are needed. Furthermore, the most significant bit in $\dot{z}$ and $\ddot{z}$ can be handled by separate logic to select from the two possible choices for z. The negative of m_i in an $n+1$-bit twos complement system has the same representation as $2^n - m_i$ for the first n bits of interest; this is why the procedure can be interpreted as adding $2^n - m_i$ if the result of the first addition is greater than $m_i - 1$. A block diagram of this type of RNS adder is shown in Figure 3.14. The adder can be further simplified by noting that if m_i is fixed, then the second adder can be customized to reduce its size. The second layer of full adders can be replaced by the equivalent of half adders since one addend is always known.

An RNS adder constructed like Figure 3.14 requires two n-bit adders, an OR gate, and n 1-bit multiplexors. The area of this adder is given by

$$A_{RNSadd} = 2nA_{fa} + A_{or} + nA_{mux}, \tag{3.37}$$

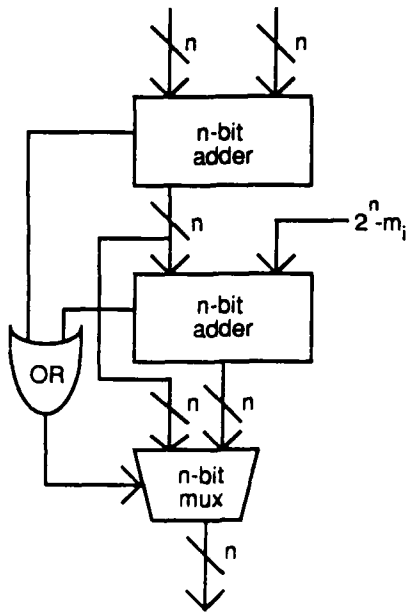


Figure 3.14: Modulo m_i adder using two binary adders.

where

$$\begin{aligned} A_{fa} &= \text{area of a full adder} \\ A_o &= \text{area of an OR gate} \\ A_{mx} &= \text{area of a multiplexor.} \end{aligned}$$

Assuming $A_{fa} \gg A_o, A_{mx}$, the size of the adder can be approximated by

$$A_{RNSadd} \approx 2nA_{fa}. \tag{3.38}$$

The time required to perform the addition is given by

$$T_{RNSadd} = (n+1)T_{fa} + T_o + T_{mx}, \tag{3.39}$$

where

$$\begin{aligned} T_{fa} &= \text{delay through a full adder} \\ T_o &= \text{delay through an OR gate} \\ T_{mx} &= \text{delay through a multiplexor.} \end{aligned}$$

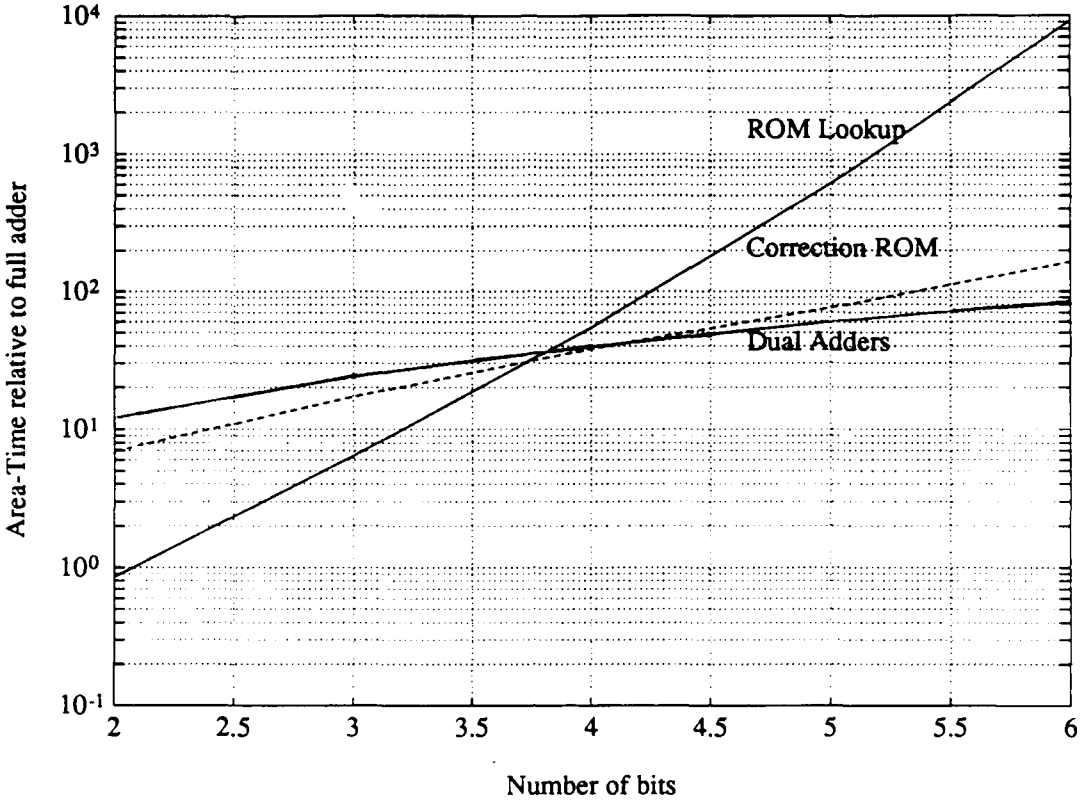


Figure 3.15: Area-time products for three modular adders.

Because data ripples through the two layers of full adders diagonally, not downward through each layer, the delay through the full adder section is equal to the number of full adders across plus the number of layers after the first one. Now, assuming $T_{fa} \gg T_o, T_{mx}$, the delay through the adder can be approximated by

$$T_{RNSadd} \approx (n+1)T_{fa}. \tag{3.40}$$

The area-time product for a modulo-m_i adder, where $2^{n-1} < m_i \leq 2^n$, can now be approximated by

$$AT_{RNSadd} = 2n(n+1)A_{RNSadd}T_{RNSadd}. \tag{3.41}$$

The area-time products for the three types of modular adders are plotted together in Figure 3.15. Each of these adders has a region in which it is superior to the other two. The adders using ROMs are better than the one that uses dual adders for moduli less than 2^3. Moduli larger than 2^3 require at least 4 bits, in which case the adder

using dual n-bit adders dominates. Since the dual adder type dominates for moduli of at least 4 bits and is almost equal to the other types of adders for 3-bit moduli, it will be the one used for comparisons against integer adders.

3.2.2 RNS Multipliers

There are several methods of performing RNS multiplication that have been discussed in the literature, some of which were explained in Chapter 1. The first and simplest method is to use a ROM look-up table; the structure would be identical to the ROM adder of Figure 3.12 except that it would be programmed for multiplication instead of addition. This type of multiplier has the advantage of running as fast as the ROM's access time, but the ROMs take up a large amount of area for larger moduli. If the modulus has n bits, the address of the ROM will contain $2n$ bits. Moduli of 5 bits (no larger than 32) require a ROM of 1K$\times$ 5 bits. Each additional bit in the moduli *quadruples* the size of the required ROM.

To alleviate this problem, two other types of multipliers have been proposed. The first, discussed by Jullien, works for prime moduli m_i and uses index calculus as shown in Figure 3.16. A one-to-one "logarithmic" mapping is defined between $\{g_n\} = \{1 \ldots (m_i - 1)\}$ and $\{k_n\} = \{0 \ldots (m_i - 2)\}$ via a primitive root α such that

$$g_n = |\alpha^{k_n}|_{m_i}. \tag{3.42}$$

Multiplication of the residues g_n corresponds to addition of the exponents k_n modulo $m_i - 1$:

$$|g_n g_j|_{m_i} = \left|\alpha^{|k_n + k_j|_{m_i - 1}}\right|_{m_i}.$$

Multiplication is done by looking up the two indices of the multiplicands in ROMs, adding them modulo $m_i - 1$, and then using a ROM to get the product from the resulting index [8]. Additional circuitry is needed to detect a zero since it has no index. This procedure is analogous to performing multiplication using logarithms. The hardware requirements are three ROMs of $2^n \times n$ bits and an n-bit modular

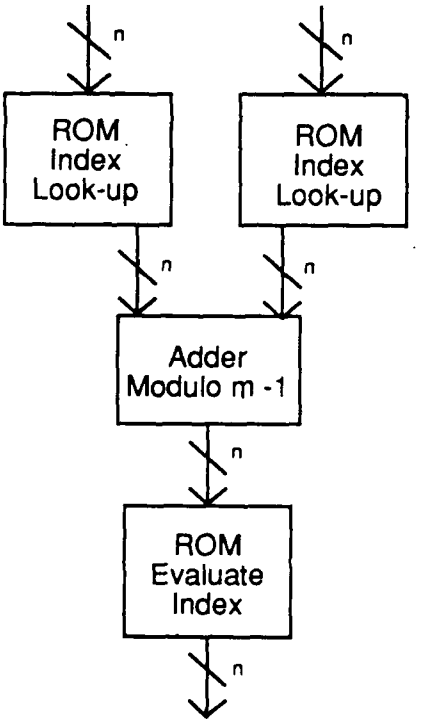


Figure 3.16: Modulo m_i multiplier using index calculus.

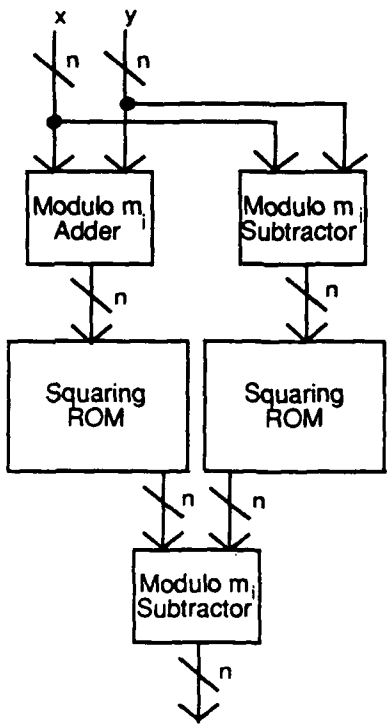


Figure 3.17: Modulo m_i multiplier using quarter squares identity.

adder; the time required for the process is equal to twice the delay through one ROM plus the delay through one adder.

Another modular multiplier which reduces ROM sizes is the quarter squares multiplier presented by Soderstrand and Vernia and shown in Figure 3.17; it takes advantage of the quarter squares identity mentioned in Chapter 1 [13]:

$$xy = \frac{(x+y)^2 - (x-y)^2}{4}. \tag{3.43}$$

Because the multiplicative inverse of 4 does not exist for even moduli, this exact procedure cannot be used for even moduli. However, Taylor has shown how this multiplier can be modified for even moduli [15]. This usually would not pose a problem anyway because the best choice for an even modulus among a set of moduli would be 2^n, so that conventional binary circuitry could be used and carries beyond n bits disregarded. The hardware cost of the quarter squares approach is three n-bit modular adders and two ROMs of $2^n \times n$ bits; the time required for the calculation

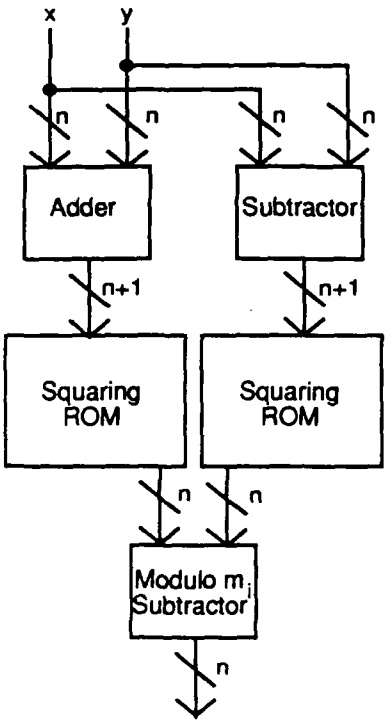


Figure 3.18: Modified modulo m_i multiplier using quarter squares identity.

is twice the time for a modular adder plus the delay through a ROM. The quarter squares multiplier takes approximately the same amount of time as the index calculus multiplier but reduces the size by one ROM in exchange for two more n-bit adders. Also, its use is not restricted to prime moduli.

Alternatively, the quarter squares multiplier could be implemented as shown in Figure 3.18. The initial addition and subtraction are performed with single n-bit adders and correction ROMs, where the ROMs are programmed to correct the sum/difference, square it, and divide by 4. This method has a possible advantage because it eliminates two adders in exchange for an additional input bit in each of the two ROMs.

A fourth way of doing modular multiplication is to multiply each bit of the multiplier by the multiplicand and then add the partial products with their proper weightings by powers of 2. This is the same algorithm used in the most simple integer

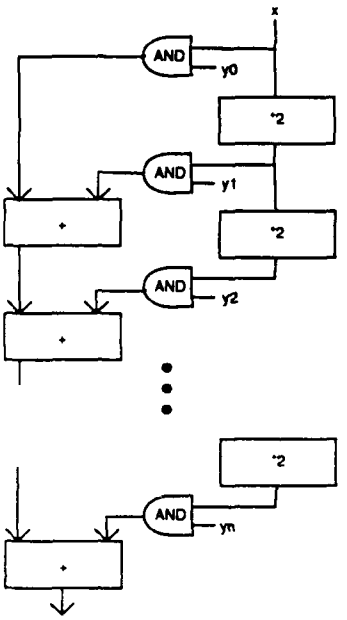


Figure 3.19: Modulo m_i multiplier using modular adders.

multipliers. A diagram of this adder is shown in Figure 3.19. The size of this multiplier is given by

$$\begin{aligned} A_{FA-mult} &= 3(n-1)nA_{fa} + 2(n-1)A_{or} + 2(n-1)nA_{mux} + n^2A_{and} \\ &\approx 3(n-1)nA_{fa}. \end{aligned} \tag{3.44}$$

Since all but the first multiplication of x by 2 is done simultaneously with the additions, the time required to complete the operation is

$$\begin{aligned} T_{FA-mult} &= nT_{fa} + (n-1)((n+1)T_{fa} + T_{or} + T_{mux}) \\ &\approx (n^2+n-1)T_{fa} \end{aligned} \tag{3.45}$$

The area-time products for these four multipliers are plotted together in Figure 3.20. The simple ROM look-up table is best for moduli of 4 bits or less, but grows exponentially worse for larger moduli. The index multiplier appears to be the best for moduli of 5–8 bits, but it has the disadvantage of only working for prime moduli. The next best multiplier for moduli of 5–8 bits is the quarter squares multiplier. The quarter squares multiplier and modified quarter squares multiplier are

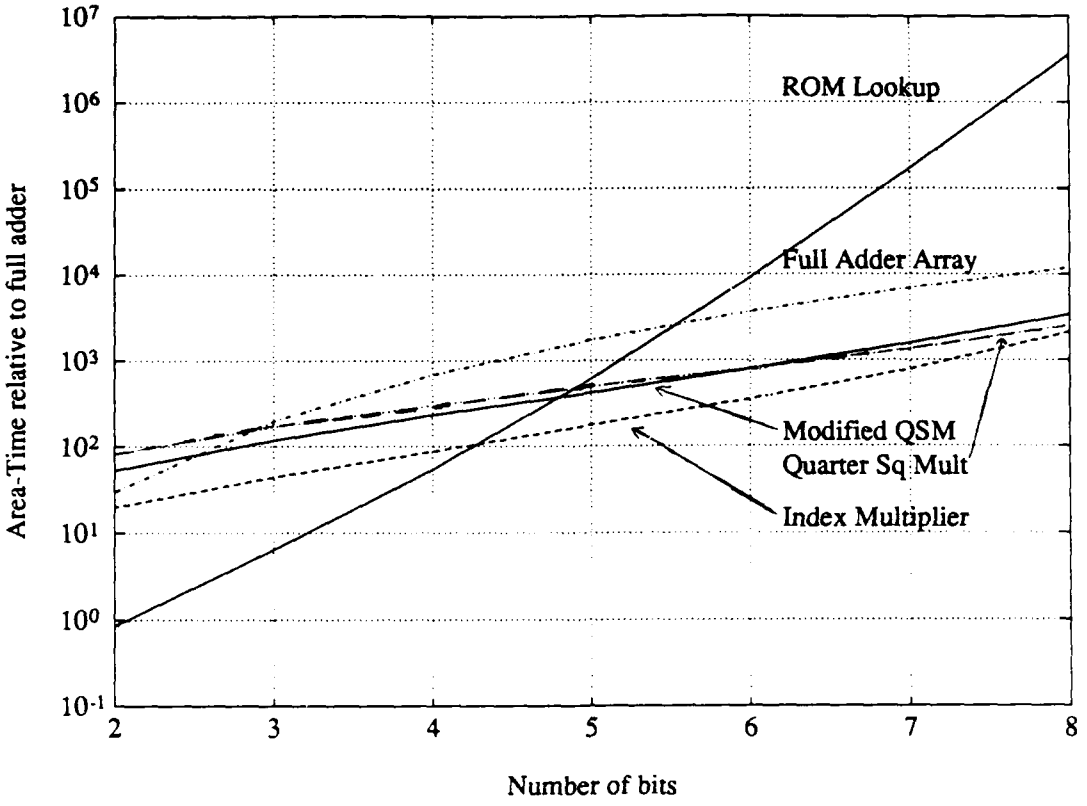


Figure 3.20: Area-time products for four modular multipliers.

close, but the modified approach is better in the 2–6 bit range, which is the more common size of moduli. For 4-bit moduli, the modified quarter squares multiplier is close in performance to the ROM look-up; smaller moduli of 2 or 3 bits are seldom used, and if they were used in parallel with larger moduli, they would have a relatively insignificant contribution to the total area-time product. For this reason, the modified quarter squares multiplier will be considered the preferred modular multiplier and will be used in comparing RNS to conventional arithmetic.

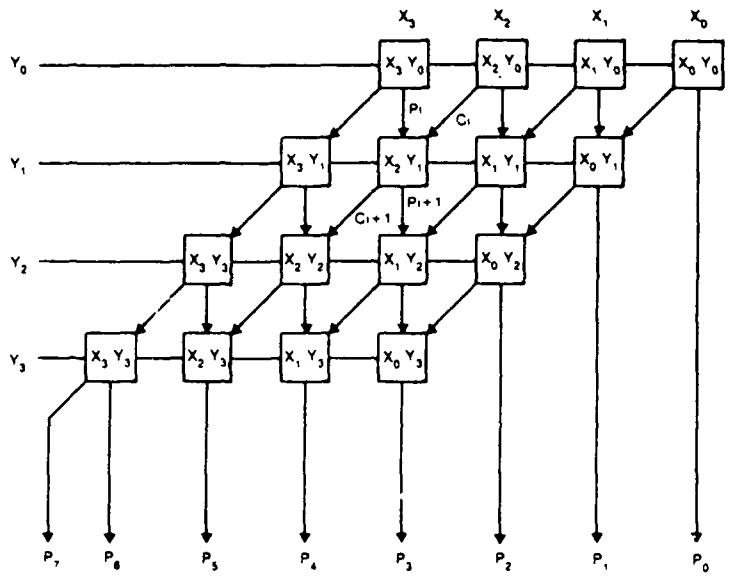


Figure 3.21: Integer multiplier.

3.3 Conventional Computational Elements

3.3.1 Adders

A conventional n-bit binary integer adder would require n full adders and have a delay of n times the delay of each full adder, for an area-time product of

$$n^2 A_{fa} T_{fa}, \tag{3.46}$$

where A_{fa} and T_{fa} are as defined above.

3.3.2 Multipliers

A simple integer multiplier is made of an array of full adders as shown in Figure 3.21. Each row of adders adds another partial product into the result, with carries propagating diagonally so that they are "saved" and added in at the following row. Not all of the adders in Figure 3.21 are actually full adders. The first row and the first

adder in the last row are half adders. The total size of the multiplier is given by

$$\begin{aligned} A_{mult} &= n(n-2)A_{fa} + \frac{n}{2}A_{fa} + n^2 A_{and} \\ &\approx (n^2 - \frac{3}{2}n)A_{fa}. \end{aligned} \tag{3.47}$$

The time to compute the product is dominated by the delay through the adders. Since carries are saved, the delay is equal to the number of layers plus the time required to add in all the carries after the final partial product is computed:

$$T_{mult} = 2nT_{fa}. \tag{3.48}$$

3.4 Side-By-Side Comparison

In a real RNS application, several modular adders would be working in parallel to replace a larger integer adder. To compare the area-time products for these two alternatives, the dynamic range for the integer adder must be broken into a product of smaller ranges for each of the moduli.

Let b be the number of bits in the integer so that it has a dynamic range of 2^b. If l moduli are used to perform the same addition with l modular adders, the ideal size of each modulus would be $2^{(\frac{b}{l})}$, or b/l bits. Of course, this is not really possible because the l moduli must be relatively prime integers, which is generally not the case if this simple formula is used; however, this formula provides a simple expression for determining the best possible moduli given l. The following three examples demonstrate the problem of picking the moduli. For the first, let $b = 8$ and $l = 3$. We desire 3 moduli with approximately $8/3 \approx 2.67$ bits per moduli. Since moduli must be integers, the largest modulus must have at least 3 bits. In this case, the moduli set {8,7,5} will work. The dynamic range is $8 \cdot 7 \cdot 5 \cdot = 280$. The total number of bits required for the moduli is $3+3+3 = 9$, 1 more bit than for the integer equivalent. For the second example, let $b = 16$ and $l = 3$. The largest modulus will have 16/3 bits, so the set {64,63,17} is chosen. The dynamic range is 68,544. This

time, $6+6+5=17$ bits are needed to encode 16-bit integers. The last example uses the same dynamic range as the previous example, except that five moduli will be used with the intention of increasing computational speed by using smaller moduli. We need approximately $16/5 = 3.2$ bits per moduli. The set {16,15,13,11,3} will work in this case. Now, however, the total number of bits required is $4+4+4+4+2=18$, resulting in 2 extra bits.

3.4.1 Adders

With b/l bits per moduli, the RNS adder will have l modular adders, each of which has an area-time product of $2\frac{b}{l}(\frac{b}{l}+1)A_{fa}T_{fa}$. The larger integer adder has an area-time product of $b^2A_{fa}T_{fa}$, which means the ratio of the RNS area-time to the integer area-time is

$$\frac{2(b+l)}{bl}. \tag{3.49}$$

RNS will have a performance advantage if this ratio is less than 1, which holds if

$$l > \frac{2b}{b-2}. \tag{3.50}$$

In order for this inequality to hold for even the largest dynamic ranges, there must be at least 3 moduli. The following section will develop results showing the performance advantage in using RNS for multiplication.

3.4.2 Multipliers

The RNS multiplier used for comparison is the modified quarter squares multiplier. There will be a multiplier for each of the l moduli of b/l bits, for a total area of l times the area of a $\frac{b}{l}$-bit multiplier and speed of one $\frac{b}{l}$-bit multiplier.

Because the area-time product for the RNS multiplier is much more complicated than that for the adder, a simple expression illustrating its performance advantage over conventional arithmetic is not possible. Instead, Figure 3.22 shows the ratio of the RNS multiplier's area-time product to the conventional integer's area-time

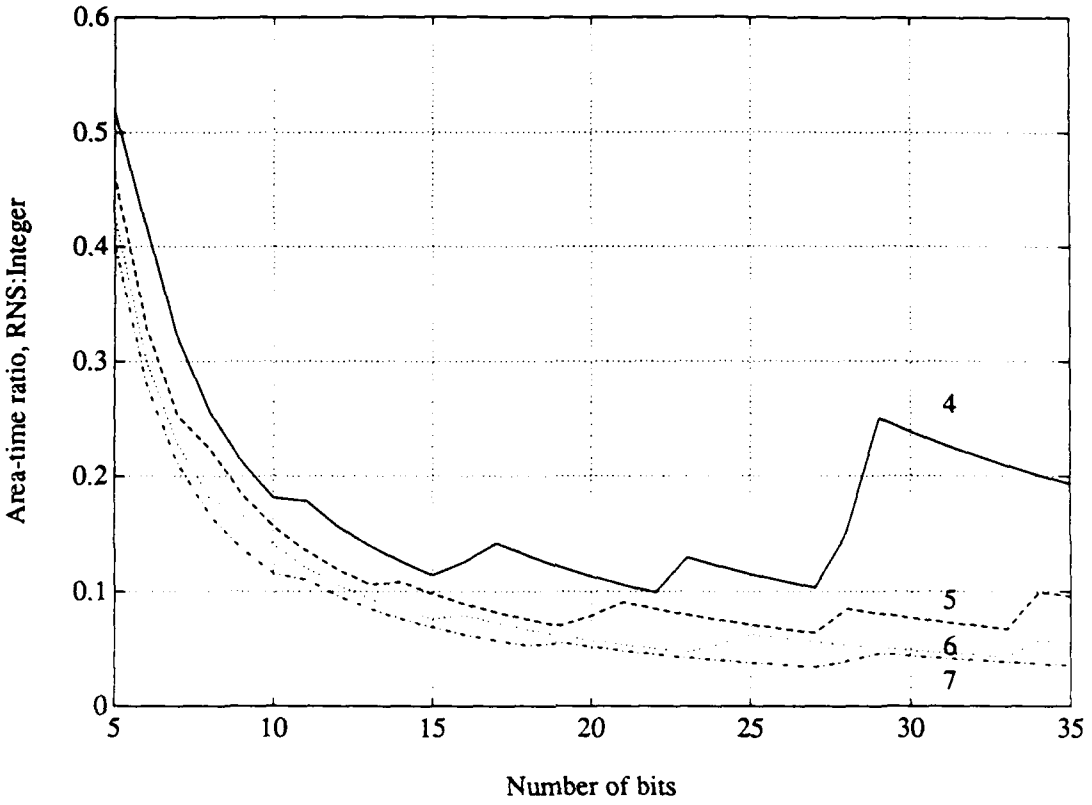


Figure 3.22: RNS/Integer multiplier area-time ratio for different numbers of moduli.

product. Each curve plots the ratio for a different number of moduli. This ratio can be as low as 1/20 for large multipliers with many moduli.

3.4.3 Registers

Registers are not computational components, but because they are used so commonly to store data temporarily and to pipeline systems both within and between components, their implementation should be examined. For the most part, registers will be almost identical for integer and RNS systems that have the same dynamic range. The real difference is that RNS will require one or two more bits in each register to match the dynamic range of an integer system. This is apparent if one looks back to Section 3.4 in which it was stated that a dynamic range of 2^b cannot truly be factored into l moduli all of size $2^{(\frac{b}{l})}$. It is also impossible to make the moduli set be $\{2^{b_1}, 2^{b_2}, \ldots, 2^{b_l}\}$ such that $\sum_{i=1}^{l} b_i = b$ because only one of the moduli may be of the

form 2^b if they are to be relatively prime. The other moduli must be prime numbers with no common factors. Thus, the actual RNS dynamic range is less than the sum of the bits required to represent all of the residues, and larger moduli with more bits are needed to realize the b-bit dynamic range of the integer system. In practice, this usually amounts to one or two more bits.

Chapter 4

DFT Algorithms

The problem of computing DFTs, for which the application of RNS arithmetic is being analyzed, will now be discussed. As explained in Chapter 1 there are several ways of computing Discrete Fourier Transforms (DFTs), all of which use some divide-and-conquer technique to break the problem into smaller pieces. The DFT is defined in Oppenheim and Schaefer [10] as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \tag{4.1}$$

where $W_N = e^{-j\frac{2\pi}{N}}$. A direct computation of equation 4.1 requires N multiplies and $N-1$ additions for each of the N output points, for a total operation count proportional to N^2. Although the equation can be manipulated for real sequences to eliminate trivial operations, the operations are generally complex, requiring several computations on the real and imaginary parts of the operands. The algorithms discussed in this chapter all attempt to reduce the number of operations required to compute the DFT by taking advantage of the periodic nature of equation 4.1. It will be shown that the Winograd Fourier Transform Algorithm is the most efficient for computing DFTs in the residue number system, and that RNS arithmetic has the greatest comparative advantage, if any, over radix-2 arithmetic for this algorithm.

4.1 Cooley-Tukey FFT

Burrus and Parks [3] explain the derivation of the many DFT algorithms, including the popular Cooley-Tukey Fast Fourier Transform (FFT). The length of an FFT should ideally be a highly composite number, with the most common FFTs having a length of the form b^m, where b is usually 2. The FFT breaks the transform into smaller pieces by factoring the length N into its factors. Each layer of an FFT removes one more factor from the original length. If the length N is factored into N_1 and N_2, then the following mapping is made for the time and frequency indices:

$$n = N_2 n_1 + n_2 \tag{4.2}$$

$$k = k_1 + N_1 k_2, \tag{4.3}$$

such that

$$\hat{x}[n_1, n_2] = x[n] \tag{4.4}$$

$$\hat{X}[k_1, k_2] = X[k]. \tag{4.5}$$

The DFT of Equation 4.1 can now be written as

$$\hat{X}[k_1, k_2] = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} \hat{x}[n_1, n_2] W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2}. \tag{4.6}$$

Examination of Equation 4.6 reveals that first N_1-point DFTs are performed for the N_2 values of n_2 in the inner summation. Then, each of the resulting points is multiplied by a twiddle factor, the term $W_N^{n_2 k_1}$. The outer summation is an N_2-point DFT which must be computed for each of the N_1 values of k_1. Since FFTs are usually performed on sequences whose lengths are a power of 2, N_1 is usually 2 and $N_2 = N/2$. The N_2-length DFTs are performed by doing another mapping and removing another factor of 2. Thus, all DFTs are reduced to 2-point DFTs, called butterflies, which are simply the sum and difference of two data points[3].

The resulting flowgraph for an 8-point FFT is shown in Figure 4.1. Because of the index mapping, the output points are in bit-reversed order. This algorithm is called

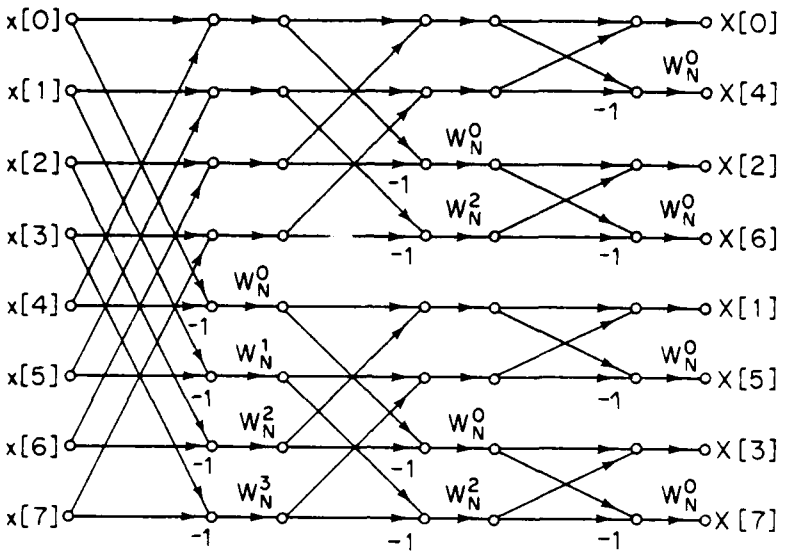


Figure 4.1: Flowgraph of an 8-point FFT [10], p. 602.

the decimation-in-frequency FFT. If N_2 were chosen to be 2 so that $N_1 = N/2$, the flowgraph would have evolved into the decimation-in-time FFT, in which the input points are in bit-reversed order.

The Cooley-Tukey FFT greatly reduces the number of operations required to compute the DFT from the number required in straightforward evaluation of Equation 4.1. If the length of the DFT is $N = 2^m$, then there will be m layers in the FFT flowgraph. Each layer requires N complex additions (or subtractions) and $N/2$ complex multiplies, as seen in Figure 4.1. Since $m = \log_2 N$, the total operation count is $N \log_2 N$ complex additions and $\frac{N}{2} \log_2 N$ complex multiplies. (Actually, the last layer requires no layers of multiplies so that the number of multiplies is $\frac{N}{2}(\log_2 N - 1) = \frac{N}{2} \log_2 \frac{N}{2}$. The expression $\frac{N}{2} \log_2 N$ is generally used for the number of multiplies because it assumes a system that is not designed to detect this irregularity.)

Implementing the FFT with integer arithmetic requires a fair amount of scaling of the data. Each layer has one add and one multiply, which continually increase the dynamic range required by the system if the numbers are not scaled. An addition doubles the dynamic range, or adds one bit. The twiddle factors are complex expo-

nentials, which reduce to sines and cosines. Ideally, these multiplies do not change the magnitude of the points; however, with integer arithmetic, the twiddle factors must be scaled up by a factor of 2 for each bit of resolution desired. Of course, in two's complement notation these extra bits can be immediately removed after the multiplication by shifting the data an equal number of bits to the right, since each shift corresponds to a division by 2. The extra bits may be discarded (truncated) or used to round the least significant bit.

Scaling is not so trivial with RNS arithmetic because it is not a weighted number system, as discussed in Chapter 2. It was shown that the number of operations required to scale each number is on the order of $2L$, where L is the number of moduli. Because of the time required to scale in the RNS, it should be avoided whenever possible. A larger dynamic range with respect to the size of the data words will reduce the amount of scaling that must be done while performing the transform. If we let b_d represent the number of bits in the data and let b_c represent the number of bits in the twiddle factors, or coefficients, then the range will grow by $b_c + 1$ in each stage of the FFT. After the first stage, the required dynamic range is $b_d + b_c + 1$ bits. The moduli chosen in an RNS implementation of the FFT must yield a dynamic range at least this large in order to allow the completion of one addition and multiplication on a pair of data points before scaling them back to the original range. If the dynamic range were increased to $b_d + 2b_c + 2$ bits, then two stages of the FFT may be completed between scaling points.

4.2 Prime Factor Algorithm

The prime factor algorithm (PFA) for the DFT is also discussed by Burrus and Parks. Developed by Good, Thomas, and Winograd, this algorithm also maps a sequence into a multi-dimensional array so that smaller DFTs may be performed along each of these dimensions. For the PFA, however, the transform length is broken into relatively

prime factors. These factors will be the lengths of the dimensions along which the smaller DFTs will be performed. The length is usually chosen so that the factors fall into a set of lengths for which Winograd has developed a set of small-N DFT algorithms which minimize the number of multiplies. These small DFTs are usually primes or powers of primes, the most common being 3, 5, 7, 9, and powers of 2 up to 16. Other lengths are possible but are not as efficient. Algorithms for longer lengths become exponentially longer and more difficult to derive [3].

The PFA mappings for n and k involve modular arithmetic. If the length N is factored into N_1 and N_2, the mappings, given by Burrus and Parks, are

$$n = |N_2 n_1 + N_1 n_2|_N \tag{4.7}$$

$$k = \left| |N_2^{-1}|_{N_1} N_2 k_1 + |N_1^{-1}|_{N_2} N_1 k_2 \right|_N . \tag{4.8}$$

Note that $k_i = |k|_{N_i}$, and that equation 4.8 is the Chinese Remainder Theorem. The mappings for n and k may also be reversed. These mappings load and unload the points along extended diagonals of a two-dimensional array. The original DFT equation now becomes [3]

$$\hat{X}(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} \hat{x}(n_1, n_2) W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2}. \tag{4.9}$$

This is a pure two-dimensional DFT; each of the summations is an independent DFT, the order of summation can be interchanged, and there are no twiddle factors. For mappings into more than two dimensions, N_1 or N_2, or both, may be further factored and the PFA used to break the N_1- and N_2-point DFTs into smaller pieces. A diagram of the PFA is shown in Figure 4.2 for a 15-point DFT. The sets of "row" and "column" DFTs are done using Winograd's algorithms, which consist of a set of additions, a set of multiplications, followed by another set of additions.

The operation count for the PFA can be found by totalling the number of operations required in each stage. The number of operations required for each of the small length-N_i DFTs is shown in Table 4.1. If the transform length is factored into

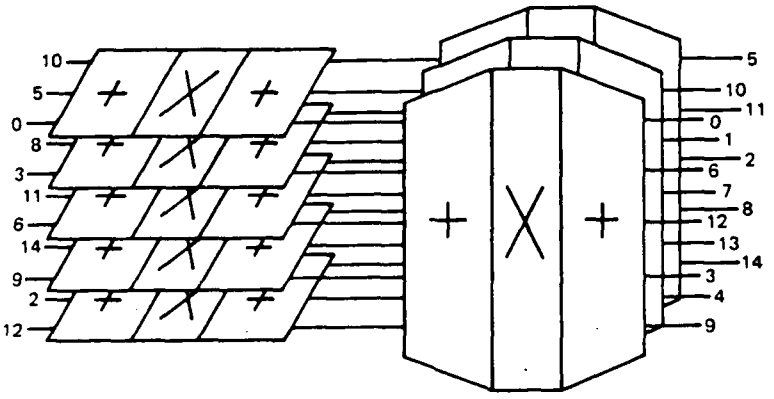


Figure 4.2: Two-factor prime factor algorithm [3],p. 63

Table 4.1: Number of real operations required for length-N DFT of real data (double for complex data)[3].

N	Total Multiplies	Multiplies By One	Adds
2	2	2	2
3	3	1	6
4	4	4	8
5	6	1	17
7	9	1	36
8	8	6	26
9	11	1	43
16	18	8	74

J dimensions, such that $N_1 \cdot N_2 \cdot \ldots \cdot N_J = N$, then there will be N/N_i N_i-point DFTs for $i = 1, \ldots, J$. If A_i and M_i represent, respectively, the number of adds and multiplies for the length-N_i DFT, the total numbers of adds and multiplies are given by

$$A = \sum_{i=1}^{J} \frac{N}{N_i} A_i \tag{4.10}$$

$$M = \sum_{i=1}^{J} \frac{N}{N_i} M_i. \tag{4.11}$$

The scaling problem exists for the PFA just as it does for the FFT. The adds and multiplies within each of the layers of DFTs increase the range of the data. In order to determine how much the range expands, more must be known about the small DFT algorithms. As explained above, there is first a layer of additions, called a preweave, that usually leads to a small data expansion. This is why the additions in the column DFTs of Figure 4.2 are represented by a trapezoidal block. Next there is a single layer of multiplies, in which each of the pieces of data out of the first stage is multiplied by a coefficient that is a combination of twiddle factors. This is followed by another set of additions, called the postweave, that yields the original number of data points. Normally, the data grows by a factor equal to the transform length in a DFT. An N_i-point transform should therefore add $\log_2 N_i$ bits onto the range of the data. If the coefficients in the multiply stage are scaled up to integers by multiplying them by 2^{b_c}, then the data will grow by $\log_2 N_i + b_c$ bits in the ith stage of the PFA. Again, the alternatives available for reducing the number of scaling operations are to either start with a larger dynamic range or to use coefficients with fewer bits, which decreases accuracy.

4.3 Winograd Fourier Transform Algorithm

The Winograd Fourier Transform Algorithm (WFTA), not to be confused with Winograd's algorithms for computing short DFTs, goes one step beyond the PFA by nesting

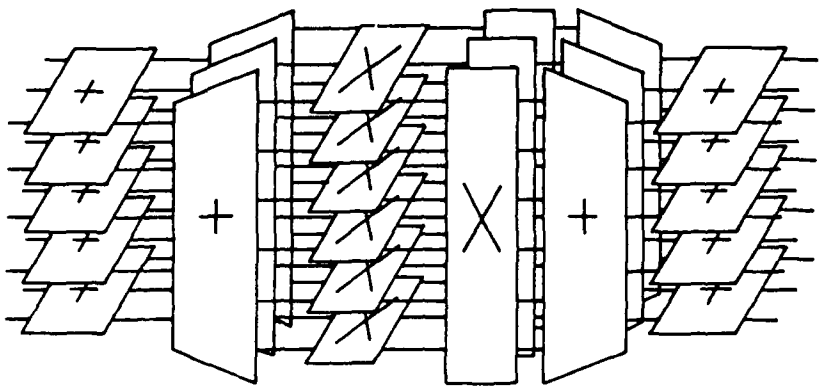


Figure 4.3: A 15-point WFTA [3],p. 71.

the three layers of operations required within each of the short DFTs. The same input and output mappings used for the PFA are also used in the WFTA. Taylor describes how the preweave, multiply, and postweave stages of the small DFTs can be viewed as matrix operations on a vector of points. Letting $\bar{x}'$ and $\bar{X}'$ be vectors of the reordered input and output points, respectively, a small-N_i DFT can be written as

$$\bar{X}' = S_{N_i} C_{N_i} T_{N_i} \bar{x}', \tag{4.12}$$

where T_{N_i} and S_{N_i} are incidence matrices (i.e., they contain small integers or fractions, only ± 1 and 0 for very short DFT lengths), and C_{N_i} contains elements only along its diagonals. The WFTA rearranges these operators so that the data points first pass through all the preweave layers of additions, then through all the multiplication stages, and finally through the postweave stages[17]:

$$\bar{X}' = (S_{N_J} * \cdots * S_{N_2} S_{N_1})(C_{N_J} * \cdots * C_{N_2} C_{N_1})(T_{N_J} * \cdots * T_{N_2} T_{N_1})\bar{x}'. \tag{4.13}$$

A diagram of this procedure for the 15-point DFT is shown in Figure 4.3. Since the multiplication stages are each point-by-point multiplications of the data by coefficients derived from twiddle factors, the separate layers of multiplications may be combined into one set of multiplications by coefficients that can be precalculated. This reduction to one layer of multiplications is the primary advantage of the WFTA.

The operation count for the WFTA is slightly more complicated than for the FFT or the PFA, and it depends on the order of the stages. Let the length N be broken down into the J factors $N_1, \ldots, N_J$, which are the lengths of the small DFTs. We will also assume that this also represents the order of the different stages. Let A_i and M_i represent the number of adds and multiplies, respectfully, in the ith stage. M_i is also the number of intermediate points, due to data expansion, in one of the N_i-point DFTs. The total number of adds is given by

$$A = \sum_{i=1}^{J-1} \left(\prod_{k=1}^{i-1} M_k \prod_{k=i+1}^{J} N_k \right) A_i. \tag{4.14}$$

The number of multiplies is simply the product of the multiply count for each of the short DFTs:

$$M = \prod_{i=1}^{J-1} M_i. \tag{4.15}$$

Since the multiplies in the WFTA are nested together, multiplications by one in the short DFTs must be counted since the corresponding factors in the following stages are generally not one.

Scaling is not so big a problem with the WFTA as it is in the FFT and PFA because there is only one layer of multiplications by coefficients scaled up to integer values. Using the same notation as above, the total dynamic range required for the WFTA (in bits) is

$$b = b_d + b_c + \log_2 N. \tag{4.16}$$

The ideal point at which to scale the data is after the multiplication stage or after the entire transform.

4.4 Comparison of Algorithms

Table 4.2 shows the number of operations required by the three algorithms for varying DFT lengths. Implementations of the three DFT algorithms in the RNS have been compared by Taylor. Because scaling is the primary weakness of the RNS, Taylor

Table 4.2: Comparison of operation counts for DFTs of complex data.

		FFT		PFA		WFTA	
N	Factors	Multiplies	Adds	Multiplies	Adds	Multiplies	Adds
63	$9 \cdot 7$			284	1236	198	1394
64	2^6	768	1152				
120	$8 \cdot 3 \cdot 5$			460	2076	288	2076
126	$2 \cdot 9 \cdot 7$			568	2724	396	3040
128	2^7	1792	2688				
240	$16 \cdot 3 \cdot 5$			1100	4812	648	5136
252	$4 \cdot 9 \cdot 7$			1136	5952	792	6584
256	2^8	4096	6144				
504	$8 \cdot 9 \cdot 7$			2524	13164	1584	14428
512	2^9	9216	13824				
1008	$16 \cdot 9 \cdot 7$			5804	29100	3564	34416
1024	2^{10}	20480	30720				
2048	2^{11}	45056	67584				
2520	$8 \cdot 9 \cdot 7 \cdot 5$			17660	82956	9504	99068

bases his comparison of the three algorithms on the ratio of transform operations to scaling operations. He shows that because the WFTA has the smallest ratio of scaling operations to transform operations, the RNS will have the greatest comparative advantage over a two's complement system for this algorithm. In fact, if scaling were to be done after the entire WFTA and if no more processing were required after the transform, the scaling could be effectively eliminated by incorporating it into the RNS-to-two's complement conversion algorithm[17]. The advantage of the WFTA is apparent from the previous parts of this chapter and from Table 4.2 in which it is shown that the WFTA minimizes the number of multiplies. Although the RNS is generally very efficient at multiplication, these multiplications are by small constants which may each have been scaled up by approximately 6–8 bits, depending on the desired accuracy. The main disadvantages of the WFTA are that it requires a complex reordering of the data at the input and output and that it is not in-place because of the data expansion in the multiplication stage. Also, because the coefficients in the multiplication stage are determined by nesting and combining the multiplication layers in the small DFTs, a new routine must be developed for every transform length, unlike the FFT. These are the primary reasons why implementations of this algorithm are uncommon even in conventional two's complement arithmetic. However, since the objective here is to determine the relative advantage of using RNS, the WFTA is an ideal focus for comparison because it provides the greatest possible chance for the RNS to outperform two's complement arithmetic. Also, since the primary reason for using RNS would be to increase performance despite the difficulty of using unconventional arithmetic, the WFTA may be a more likely choice because of its minimal multiplication count.

Chapter 5

Performance Comparison

The performance of RNS and two's complement systems for computing DFTs will now be compared using the area-time metric. Chapter 3 developed expressions for the area-time products of the basic components in RNS and two's complement integer arithmetic. Chapter 4 discussed the possible DFT algorithms and the Winograd Fourier Transform Algorithm was found to be the most appropriate for RNS. In addition, operation counts were found for the different size WFTAs. The number of adds and multiplies required for the transform, plus the additional operations required for conversion and scaling, are now combined with the area-time measures for the individual components to determine such a measure for the two systems.

5.1 System Area-Time Products

The area-time products for the systems being compared will take into account the addition and multiplication operations only. Area-time products for the adds and multiplies can first be developed separately:

$$(AT)_{add} = (AT)_{adder} \times \#adds \tag{5.1}$$

$$(AT)_{mult} = (AT)_{multiplier} \times \#multiplies. \tag{5.2}$$

These approximations are relatively straightforward. It would take a single adder a time equal to the number of adds times the time per add to do all the adds. Alternatively, assuming the adds could be ordered properly, k adders could perform the same number of adds in $\frac{1}{k}$ times the above time, but with k times the size.

The other part of the problem is to combine $(AT)_{add}$ and $(AT)_{mult}$ into a single figure. We will add these two quantities, which reflects the area-time for the most efficient system, in which the hardware is perfectly allocated between additions and multiplications. Since the problem is not finished until both the additions and multiplications are done, we are interested in finding

$$AT = \min((A_{add} + A_{mult}) \max(T_{add}, T_{mult})). \tag{5.3}$$

Assuming that $A_{op}T_{op}$ is constant for both addition and multiplication, the total area-time, AT, will be minimized when

$$T_{add} = T_{mult}. \tag{5.4}$$

If one of these times were smaller than the other, the corresponding area could be decreased without affecting the system's time, thereby reducing AT further. The desire to make the adds and multiplies take the same amount of time so that they may be done together may seem contradictory, since the WFTA, as explained in Chapter 4, ends with a set of output additions. However, when there is a constant stream of data and DFTs to perform, making these two sets operations take equal times minimizes the average time for an efficient program and controller. When $T_{add} = T_{mult}$, Equation 5.3 reduces to

$$AT = (AT)_{add} + (AT)_{mult}. \tag{5.5}$$

5.2 Analysis of Problems

The example DFT problems were analyzed by breaking the problem into several parts. First, the size of the DFT, number of bits, and RNS system are defined. The number

of bits in the data and coefficients, b_d and b_c respectively, can be chosen depending on the resolution desired. The required number of bits for an N-point WFTA, given by Equation 4.16, is

$$b = b_d + b_c + \log_2 N. \tag{5.6}$$

The next step is to determine the number and size of the moduli for an RNS system. To show how the performance changes with the number of moduli, we will let l vary, usually from 3 to 8 moduli. A simple approximation for the size of the required moduli is $\frac{b}{l}$ bits.

With the system defined, the next step is to determine the area-time products for the modular adders and multipliers. The general formula is

$$AT_{\text{component}} = (l \cdot \text{size of component}) \cdot (\text{time for component}). \tag{5.7}$$

The same is also done for two's complement components, using the original b-bit range. The number of operations to be performed by the components will depend on the size of the DFT and on the number system. For two's complement arithmetic, the number of operations are given in Table 4.2 and can be represented by A_{DFT} and M_{DFT}. For RNS, there is also the requirement to convert and scale the data. For complex data, conversion to RNS requires one ROM lookup per point. Following the DFT, the data may be scaled or converted to two's complement, but usually not both. (If so, the scaling may be done after conversion, at which point it is essentially a "free" operation.) Either of these processes takes on the order of l adds and multiplies in each RNS channel per point. A mixed-radix conversion actually requires $l-1$ of each operation, but requires $l-1$ more after the coefficients are found. A safe assumption is that there are at least l operations, the number required for scaling. Since we have complex data, there are $2l$ adds and the same number of multiplies. The total numbers of RNS operations are given below:

$$Add_{RNS} = Add_{DFT} + 2l \cdot \#\text{ points} \tag{5.8}$$

$$Mult_{RNS} = Mult_{DFT} + 2l \cdot \#\text{ points}. \tag{5.9}$$

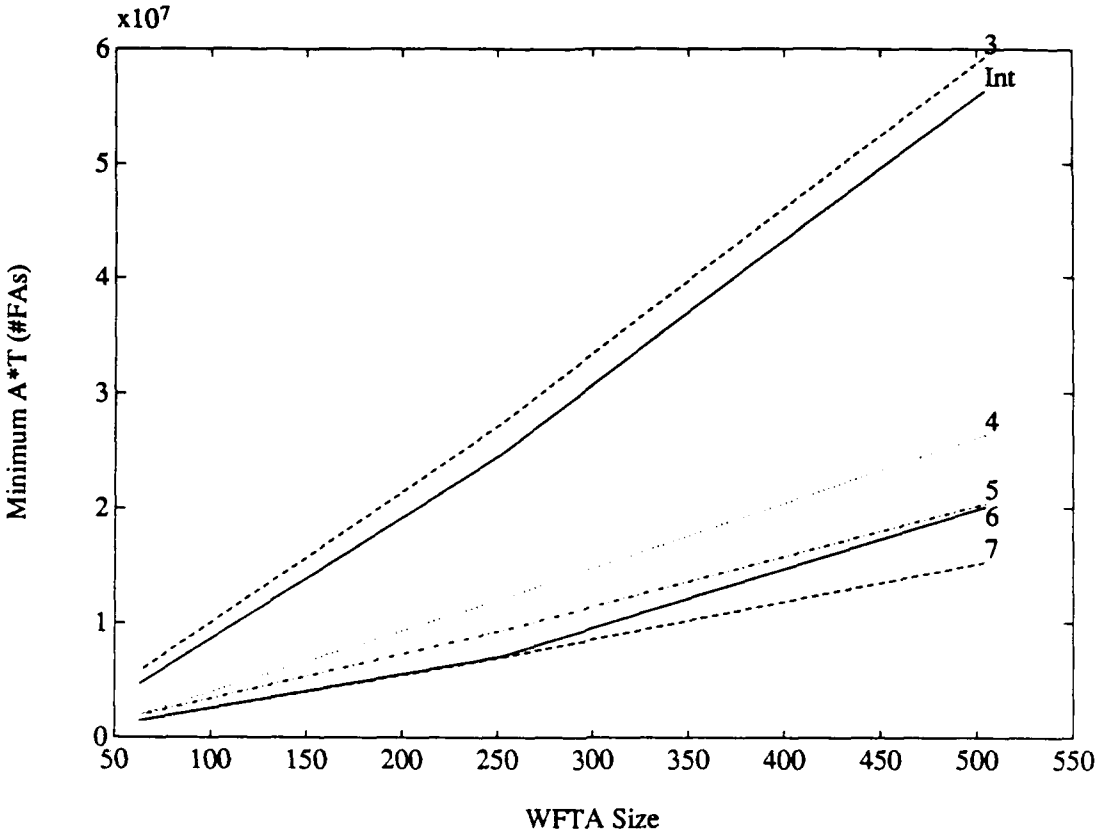


Figure 5.1: Area-time products for systems performing WFTAs.

The area-time products for the two systems can now be found by multiplying the area-time for each component by the number of corresponding operations and adding the area-time products for the two types of operations.

Results comparing RNS and integer systems are shown in Figure 5.1. These results were generated with Matlab; the script file is included in the appendix. The lines show the area-time products versus transform length for different numbers of moduli and for a two's complement integer system. The number of bits in the data, b_d, and in the coefficients, b_c, is assumed to be 8. The plot suggests that RNS can provide a significant performance advantage if at least 4 moduli are used. However, these results were generated with the ideal assumption that all of the moduli had b/l bits.

The results of Figure 5.1 may be made more realistic by changing the way the moduli are selected. In Chapter 3, it was noted that $\frac{b}{l}$ may not be an integer and that not all of the moduli can be of the same. The moduli selection may be made

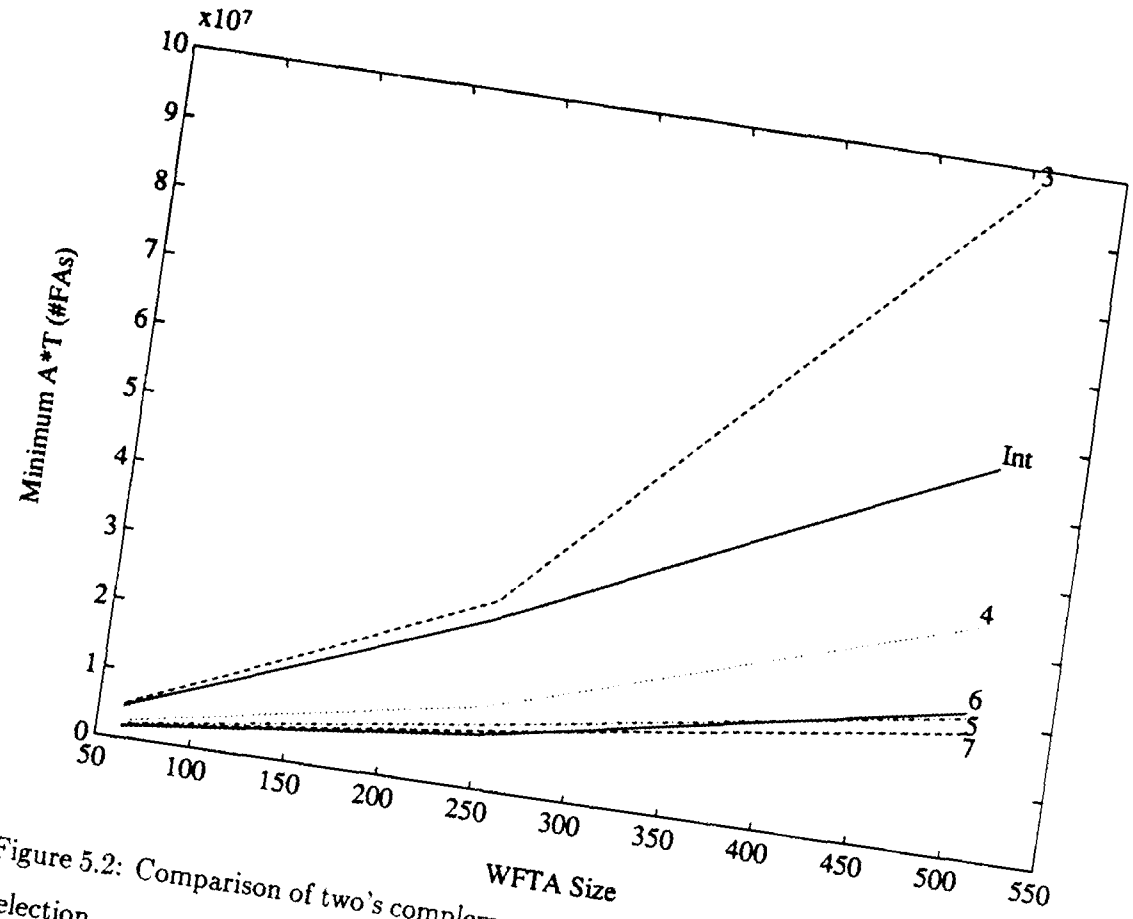


Figure 5.2: Comparison of two's complement and RNS systems using modified moduli selection.

more realistic by letting the size of the first modulus be

$$b_1 = \left\lceil \frac{b}{l} \right\rceil, \tag{5.10}$$

where $\lceil x \rceil$ represents the smallest integer greater than or equal to x. The remaining moduli are chosen using the formula

$$b_i = \left\lceil \frac{b - \sum_{j=1}^{i-1} b_j}{l - i + 1} \right\rceil. \tag{5.11}$$

Hence, some moduli will be larger than others and the time of the largest modulus will be the limiting factor. The areas of the RNS components are the sums of the areas of the components for each of the moduli, not simply l times the area of one. The results when the moduli are chosen this way are shown in Figure 5.2. The required number of bits are determined from Equation 5.6 for each transform length, and the above formulas are used to determine a moduli set for each transform length and value of l,

the number of moduli. The Matlab script file is again included in the appendix. Note that the RNS with at least 4 moduli can still outperform conventional arithmetic, but the performance advantage has been reduced. For the 504-point DFT with 7 moduli, the advantage has been reduced from a factor of 4 to a factor of 3.

We will now further restrict the selection of moduli by recognizing that not only must the moduli be different, but they may not all be of the form 2^{b_i}. The first modulus is usually chosen to be of this type, and the second to be of the form $2^{b_i} - 1$, but other moduli can not be even or contain any other factors in common with the previous moduli. This means that there is some penalty in terms of dynamic range loss from the ideal b_i-bits, as discussed in Chapter 3. This penalty is much more significant for large numbers of moduli. The actual penalty depends on the desired range and exact number of moduli, but an average provides a good approximation. Studying a few examples for 20–30 bit ranges has shown that the average penalty is about 0.2 bits for 5-moduli systems, 0.25 bits for 6 moduli, and 0.43 bits for 7 moduli. RNS moduli can be chosen to provide an adequate dynamic range for different problems, taking into account this loss; the results for this approach are shown in Figure 5.3. The maximum performance advantage has now been reduced to about a factor of 2.7.

A final plot is now included which compares the results when the two's complement system is made more realistic by recognizing that it need not have the same range as RNS because of the ability to scale quickly whenever needed. The required number of bits for a minimal amount of scaling is approximately the original number of bits, b_d, plus the maximum of b_c and $\log_2 N$. This allows for the completion of at least the preweave stage before scaling, but also provides enough dynamic range to complete the multiplication. These results are shown in Figure 5.4. Under this scheme, the integer system has approximately .87 times the area-time of the best RNS system for a 504-point DFT.

The analyses in this chapter have compared RNS to two's complement systems by calculating an area-time product based on the area-time products of the basic com-

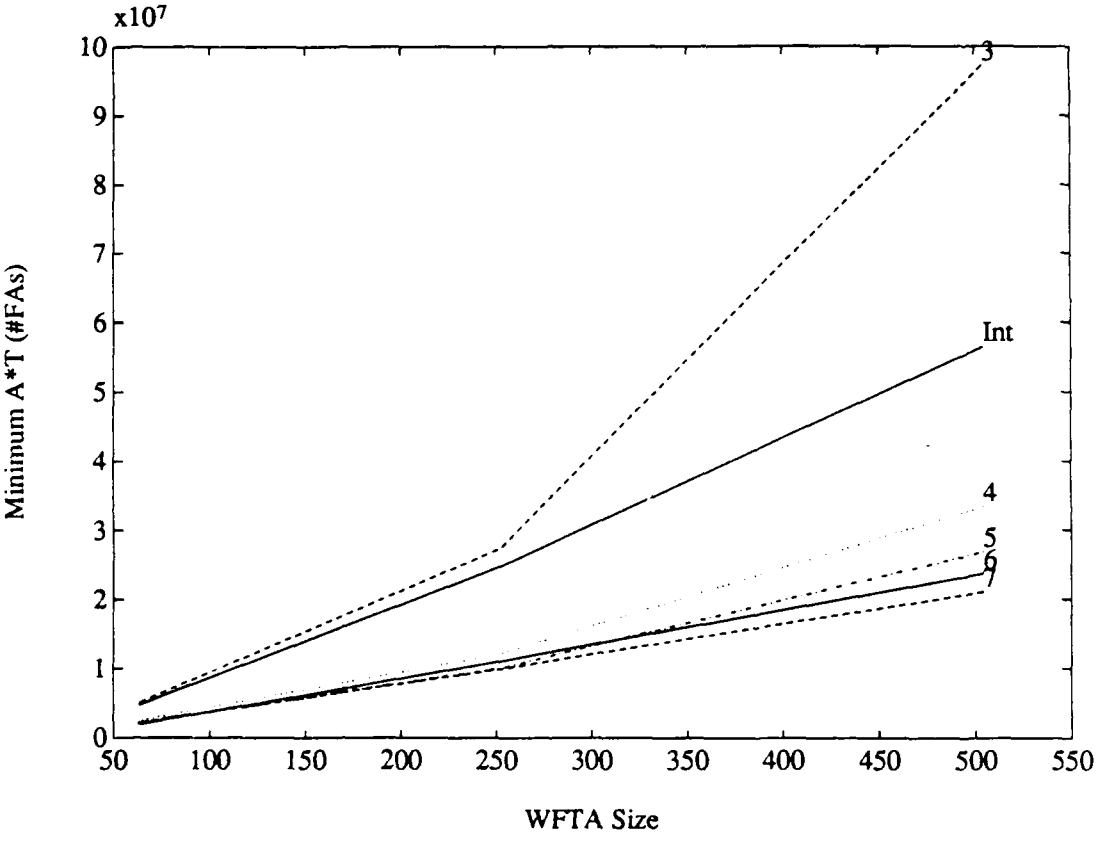


Figure 5.3: Comparison of two's complement and RNS systems using dynamic range penalty in moduli selection.

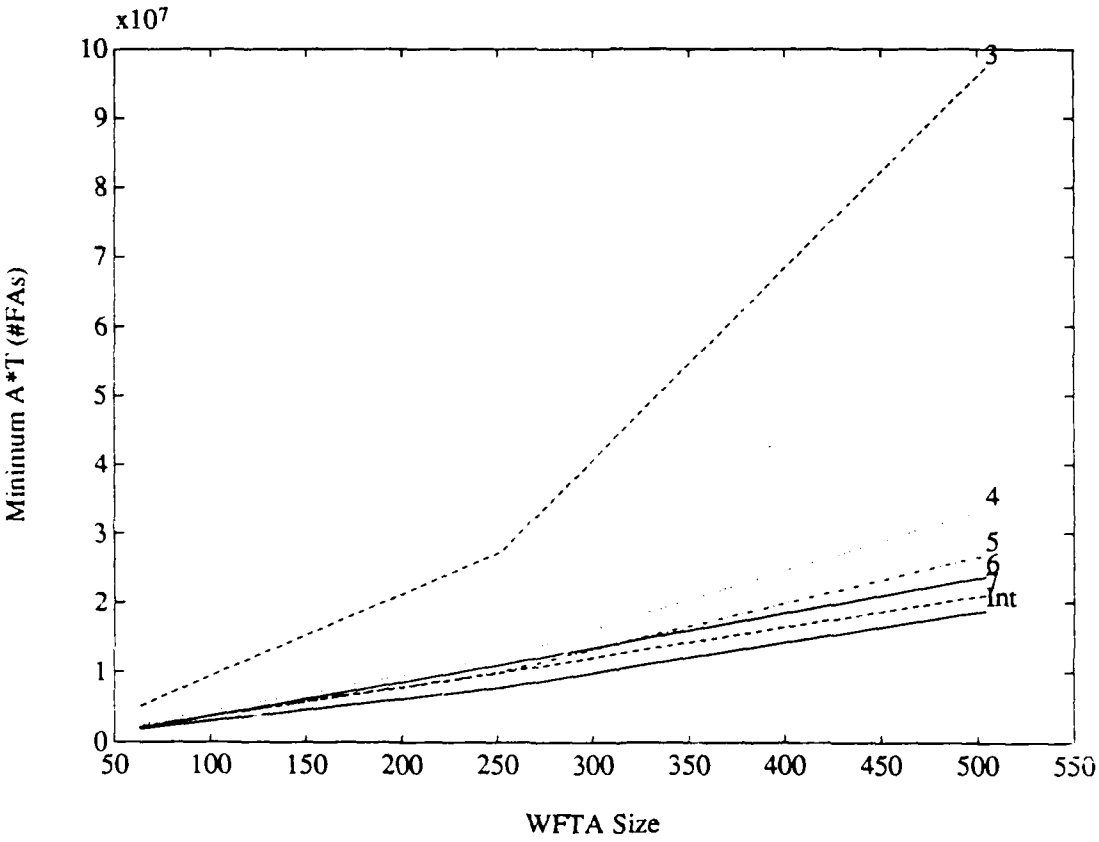


Figure 5.4: Comparison of two's complement and RNS systems using smaller dynamic range two's complement.

putational components and the numbers of the different operations. The computed values may not be obtainable in practice because there has been no accounting for required registers, controllers, and interconnect. However, as previously stated, our goal is to determine whether RNS has an advantage based purely on its efficiency in performing arithmetic. Non-computational components have not been addressed because these parts would be nearly identical in the two types of systems. While RNS is competitive with two's complement arithmetic in a strict "apples-and-apples" comparison of efficiency in performing a WFTA, RNS is the not the clear winner when realistic assumptions are made about how such a problem would really be solved in two's complement. The most important difference is that a two's complement system would not have to begin with a dynamic range large enough to solve the problem without scaling in order to successfully compute the DFT.

Chapter 6

Conclusions

6.1 Performance of RNS

The results of Chapter 5 show that RNS is not a clear winner over two's complement for the problem of performing WFTAs. RNS initially appeared to have a significant advantage, largely due to the small size and delay of the ROMs used in the quarter squares multiplier. In Chapter 3 it was shown how favorably relatively small ROMs compared to full adders in terms of size and speed. However, this advantage is outweighed by the disadvantage of not having an efficient method of scaling and by the necessity of converting data back to two's complement through a lengthy series of operations.

These results for the WFTA indicate that RNS does not provide an advantage over two's complement for the general problem of computing a DFT. In Chapter 4 it was shown that among the most efficient DFT algorithms known, the WFTA provides RNS with the greatest chance of outperforming two's complement. This is because there is only one layer of multiplies along the data path, thus minimizing the growth in dynamic range due to multiplication by constants scaled up to the desired number of bits of resolution. Since RNS does not appear to be more efficient than two's complement for WFTAs, it will not be better for other algorithms, such as the PFA

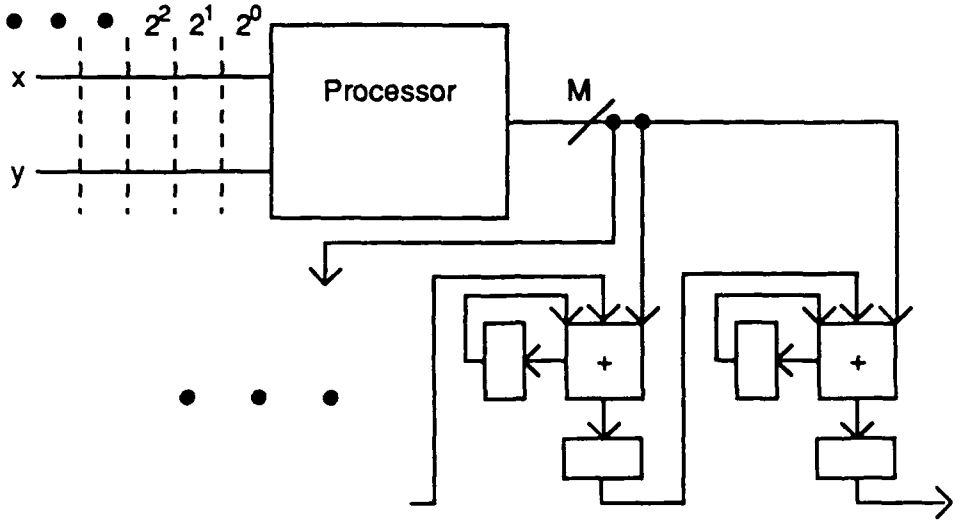


Figure 6.1: General block diagram of system using distributed arithmetic.

and FFT.

6.2 New Ideas

The size and speed of relatively small ROMs suggests that they may be used to replace computational components working on conventional two's complement arithmetic, if they can be used in situations where a small addressing space is possible. A possible application is in distributed arithmetic, in which data to be processed is fed into components in a bit-serial fashion. The components process the data and continuously shift and add in results for successive bits such that there is also a constant stream of bits coming out. These bits can be immediately used by the next component, even though the previous component is still working on the more significant bits of the same piece of data. A general example of this technique is shown in Figure 6.1.

Distributed arithmetic may be used with ROMs to implement the series of operations needed to compute a radix-2 or radix-4 butterfly used in a Cooley-Tukey FFT. A diagram of a radix-4 butterfly implemented in distributed arithmetic is shown in Figure 6.2. Here, ROMs are used to multiply three of the four outputs by precom-

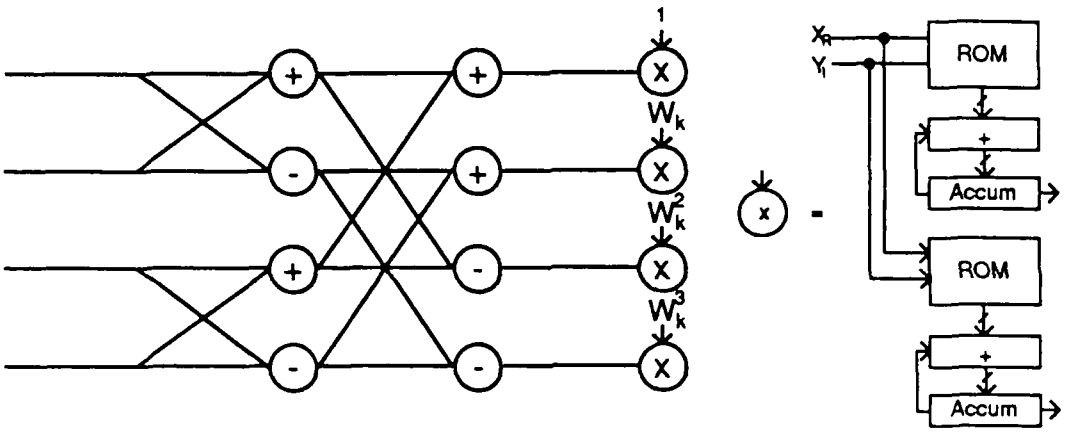


Figure 6.2: Radix-4 butterfly using distributed arithmetic.

puted twiddle factors. Since the data comes in one bit at a time, the ROMs need only contain two address lines—one for the real part and one for the imaginary. Assuming that the twiddle factors are represented by 8-bit two's complement numbers, the ROMs will output 9-bit numbers representing either the real or imaginary part of the product of the twiddle factor with the real and imaginary parts of the data represented by the two current bits. The ROMs produce 9-bit products because each part of the complex product requires a sum of two 8-bit products. Since each of the two layers of adds before the complex multiplication add another bit to the result, there is a total growth of 11 bits in the range of the data passing through the component.

The area and time of an FFT implemented by connecting cells of the form shown in Figure 6.2 into a large array can now be computed. We will calculate these measures for a 256-point FFT, a convenient size since $4^4 = 256$. The butterfly can be broken down into

- 16 full adders/registers for the 8 complex adds in the butterfly,
- $2(b_c + 1)$ registers to delay the first complex output, multiplied by 1,
- 6 ROM/accumulate systems for the results of the three complex multiplications,

each with the following:

- one $2^2 \times (b_c + 1)$-bit ROM
- a $(b_c + 1)$-bit full adder
- $2(b_c + 1)$ registers.

The area of the butterfly is

$$(16 + 6(b_c + 1))\text{full adders} + (16 + 14(b_c + 1))\text{registers} + 6(2^2 \times (b_c + 1))\text{-bit ROMs.}$$

We will assume that a 1-bit register is about the same size as a full adder and that $b_c = 8$, as before. From the formulas developed earlier, the size of a $(2^2 \times 9)$-bit ROM is about 1.12 full adders. The above expression reduces to an area of 219 full adders. A 256-point FFT using radix-4 butterflies will have $\log_4 256 = 4$ layers, each with $\frac{256}{4} = 64$ butterflies. The total area is

$$(64 \cdot 4) \times 219 = 56,064 \text{ full adders.}$$

The time estimate begins with an estimate of the clocking period. The access time for the given ROMs is approximately .45 times the delay of a full adder. The worst case delay is one ROM look-up, a one-bit add, and storage in a register. This total delay can be estimated to be the delay of 2.5 full adders, assuming the register is the same speed as a full adder. It was found above that there is a growth of 11 bits in the range of numbers passing through each butterfly. Since there are 4 layers, the total growth is 44 bits. Assuming the original data has 8 bits, it will take a total of 52 clock cycles for the last bit of data to come out. Bits 9–52 of the input will be sign extended. The total time to perform the FFT is now

$$52 \cdot 2.5 = 130 \text{ full adder delays.}$$

The area-time product is

$$56,064 \cdot 130 = 7,288,320(AT)_{FA}$$

In Chapter 5, it was found that the best area-time product for an RNS system doing a comparable WFTA (252 points) is 10,011,000. The best case is when 5 moduli are used. When the two's complement system is modified to allow a smaller dynamic range in exchange for scaling, it achieves an area-time product of 7,749,056 for the 504-point WFTA. Neither of these two measures include the cost of the registers required to connect or pipeline the computational components, yet the distributed arithmetic system, which takes registers into account, outperforms both. In addition, the distributed arithmetic system is implementing the DFT via the FFT, instead of the WFTA. The measures above account for arithmetic computations only, so the systems using the WFTA have an inherent advantage. The FFT implementation described above, however, is computationally more efficient, yet does not require the complicated reordering of the input and output points as does the WFTA. Also, if we allow rounding in intermediate stages, then the time per problem is much less because it will not be necessary to wait for 52 bits of output.

6.3 Suggestions for Further Research

A promising area for further research now is the use of distributive arithmetic for computing DFTs. The quick analysis above shows that this approach appears to have major advantages over conventional arithmetic, especially when ROMs are programmed to implement operations such as complex multiplication, which normally requires a series of operations on just a few sets of inputs. With more work, this technique may be refined to further reduce the size and time required by the system.

The RNS does not show any clear advantages over two's complement for the problem of computing DFTs. The major disadvantage is the growth in dynamic range when DFTs are calculated. This growth is worse for RNS because the twiddle factors of the DFTs must be scaled up to integer values but cannot be easily scaled back down. RNS would be more suitable for problems that deal mainly with integers and that do

not pose dynamic range problems. One application that has been considered is in the design of an address generator for a system using the PFA or the WFTA. The input and output reordering equations for these algorithms were presented in Chapter 4 and were found to be based fundamentally on modular arithmetic. The different modular components could compute the parts of multi-dimensional addresses independently for a memory oriented towards this kind of addressing. This application would also deal exclusively with integers and would not pose problems of dynamic range.

Appendix A

Matlab M-files

```
function a=arom(no)
ni=[ no' (no+ones(no))' 2*no']';
no=[1;1;1]*no;
c=round(0.5*ni-0.5*(log(no)/log(2))-.0001);
r=round(0.5*ni+0.5*(log(no)/log(2))+.0001);
%ROM layout area in lambda units
w1=16*r+27.*ones(no);
w2=9.*no.*exp(c*log(2))+2.*ones(no);
w=w1+w2;
h1=exp(r*log(2))*8+27;
h2=18*c;
h=h1+h2;
a=w.*h/7676; %scale to area of full adder
```

```
function a=aromm(ni,no)
c=round(0.5*ni-0.5*(log(no)/log(2))-.0001);
r=round(0.5*ni+0.5*(log(no)/log(2))+.0001);
%ROM layout area in lambda units
w1=16*r+27;
w2=9.*no.*exp(c*log(2))+2;
w=w1+w2;
h1=exp(r*log(2))*8+27;
h2=18*c;
h=h1+h2;
a=w.*h/7676; %scale to area of full adder
```

```
function t=trom(no)
ni=[no' (no+ones(no))' 2*no']';
no=[1;1;1]*no;
c=round(0.5*ni-0.5*(log(no)/log(2))-.0001);
r=round(0.5*ni+0.5*(log(no)/log(2))+.0001);
%ROM time in nanoseconds
trow=5.77e-4*exp(r*log(4)) +.0389*r;
trom=5.77e-4*(no.*no).*exp(c*log(4))+.0389*exp(r*log(2));
tcol=(ones(no)+2*c).*c*0.0493;
t=(trow+trom+tcol)/1.168; %scale to time full adder
```

```
function t=trom(ni,no)
c=round(0.5*ni-0.5*(log(no)/log(2))-.0001);
r=round(0.5*ni+0.5*(log(no)/log(2))+.0001);
%ROM time in nanoseconds
trow=5.77e-4*exp(r*log(4)) +.0389*r;
trom=5.77e-4*(no.*no).*exp(c*log(4))+.0389*exp(r*log(2));
tcol=(1+2*c).*c*0.0493;
t=(trow+trom+tcol)/1.168; %scale to time full adder
```

```
% layout.m
% computes area, time, and product for ROMs of various sizes
no=2:18; % # of bits (different cases)
xax=no;
ni=[ no' (no+ones(no))' (2*no)'];
cases=[ 1 ; 1; 1];
no=cases*no;
n1s=ones(no);
a=7676; % area of full adder in lamda**2
t=1.168;  % delay through full adder in nsec

c=round(0.5*ni-0.5*(log(no)/log(2))-.0001);
r=round(0.5*ni+0.5*(log(no)/log(2))+.0001);

%ROM layout area in lambda units
w1=16*r+27.*ones(no);
w2=9.*no.*exp(c*log(2))+2.*ones(no);
w=w1+w2;
h1=exp(r*log(2))*8+27;
h2=18*c;
h=h1+h2;
arom=w.*h; %scale to area of full adder

%ROM time in nanoseconds
trow=5.77e-4*exp(r*log(4)) +.0389*r;
trom=5.77e-4*(no.*no).*exp(c*log(4))+.0389*exp(r*log(2));
tcol=(n1s+2*c).*c*0.0493;
trom=(trow+trom+tcol); %scale to time full adder

atrom=arom.*trom;
axis([2,18,3,13]);
semilogy(xax,arom);
xlabel('Bits in ROM');
ylabel('Size (square lambdas)');
grid;
!del arom.met
meta arom;
pause;
```

```
axis([2,18,-1,9]);
semilogy(xax,trom);
xlabel('Bits in ROM');
ylabel('Speed (ns)');
grid;
!del trom.met
meta trom;
pause;
axis([2,18,3,21]);
semilogy(xax,atrom);
xlabel('Bits in ROM');
ylabel('Area-Time (ns*square lambdas)');
grid;
pause;

arom=arom/a;
trom=trom/t;
atrom=atrom/(a*t);
axis([2,18,-1,9]);
semilogy(xax,arom);
xlabel('Bits in ROM');
ylabel('Size (# FAs)');
grid;
!del aromr.met
meta aromr;
pause;
axis([2,18,-1,9]);
semilogy(xax,trom);
xlabel('Bits in ROM');
ylabel('Speed (# FAs)');
grid;
!del tromr.met
meta tromr;
pause;
axis([2,18,-1,15]);
semilogy(xax,atrom);
xlabel('Bits in ROM');
ylabel('Area-Time (# FAs)');
```

```
grid;
!del atromr.met
meta atromr;
```

```
% addcomp.m
% comparison of RNS adders
no=2:6;
a=arom(no);
t=trom(no);
arom=a(3,:);
aromcorr=no+a(2,:);
adualadd=2*no;
aplot=[arom' aromcorr' adualadd'];
semilogy(no,aplot);
xlabel('Number of bits');
ylabel('Area relative to full adder');
labels=['ROM Lookup    ';
        'Correction ROM';
        'Dual Adders   '];
text(4.15*ones(aplot(1,:)),aplot(4,:)+[0 0 -6], labels);
pause;
trom=t(3,:);
tromcorr=no+t(2,:);
tdualadd=no+ones(no);
tplot=[trom' tromcorr' tdualadd'];
semilogy(no,tplot);
xlabel('Number of bits');
ylabel('Time relative to full adder');
labels=['ROM Lookup    ';
        'Correction ROM';
        'Dual Adders   '];
text(4.65*ones(aplot(1,:)),tplot(5,:)+[0 0 -3], labels);
pause;
atplot=aplot.*tplot;
semilogy(no,atplot);
xlabel('Number of bits');
ylabel('Area-Time relative to full adder');
labels=['ROM Lookup    ';
        'Correction ROM';
        'Dual Adders   '];
text(4.65*ones(aplot(1,:)),atplot(5,:)+[-8000 0 -50], labels);
```

```
grid;
!del addcomp.met
meta addcomp;
```

```
% multcomp.m
% comparison of RNS multipliers
n=2:8;
a=arom(n);
t=trom(n);
arom=a(3,:);
aind=3*a(1,:)+2*n;
aqsm=6*n+2*a(1,:);
afa=3*(n-ones(n)).*n;
aqsm2=4*n+2*a(2,:);
aplot=[arom' aind' aqsm' afa' aqsm2'];
semilogy(n,aplot);
xlabel('Number of bits');
ylabel('Area relative to full adder');
labels=['ROM Lookup      ';
        'Index Multiplier';
        'Quarter Sq Mult ';
        'Full Adder Array';
        'Modified QSM    '];
text(6.1*ones(aplot(1,:)),aplot(7,:)+[0 0 0 0 0], labels);
grid;pause;
trom=t(3,:);
tind=n+ones(n)+2*t(1,:);
tqsm=2*(n+ones(n))+t(1,:);
tfa=n.*n+n-ones(n);
tqsm2=2*n+ones(n)+t(2,:);
tplot=[trom' tind' tqsm' tfa' tqsm2'];
semilogy(n,tplot);
xlabel('Number of bits');
ylabel('Time relative to full adder');
labels=['ROM Lookup      ';
        'Index Multiplier';
        'Quarter Sq Mult ';
        'Full Adder Array';
        'Modified QSM    '];
text(6.1*ones(tplot(1,:)),tplot(7,:)+[0 0 0 0 0], labels);
grid;pause;
```

```
atplot=aplot.*tplot;
semilogy(n,atplot);
xlabel('Number of bits');
ylabel('Area-Time relative to full adder');
labels=['ROM Lookup      ';
        'Index Multiplier';
        'Quarter Sq Mult ';
        'Full Adder Array';
        'Modified QSM    '];
text(6.1*ones(atplot(1,:)),[1e6 1e2 800 1e4 2e3], labels);
grid;
!del multcomp.met
meta multcomp
```

```
% rnsmcomp.m
% comparison of RNS multipliers to integer multipliers
%   for different numbers of moduli
l=4:7;
b=5:35;
l=l';
L=l*ones(b);
ratio=((ones(l)./l)*b);
for i=1:(size(l)*[1;0])
   a=arom(ratio(i,:));
   rnsa(i,:)=(4*ratio(i,:)+2*a(2,:));
   t=trom(ratio(i,:));
   rnst(i,:)=2*ratio(i,:)+ones(b)+t(2,:);
end;
rnsat=L.*rnsa.*rnst;
%rnsat=L.*(6*ratio+2*0.122*exp(ratio*log(2.03)));
%rnsat=rnsat.*(2*(ratio+ones(ratio))+0.0413*exp(ratio*log(1.93)));
B=ones(l)*b;
intat=(B.*B-1.5*B).*B*2;
atplot=rnsat./intat;
plot(b,atplot)
%semilogy(b,atplot);
grid;
ylabel('Area-time ratio, RNS:Integer');
xlabel('Number of bits');
labels=['4';
        '5';
        '6';
        '7'];
text(31*ones(atplot(:,1))',[.25 .08 .04 .01], labels);
!del rnsmcomp.met
meta rnsmcomp
```

```
% atcomp.m
% area-time comparison for DFT computation
% ideal selection of moduli
dft=[63 252 504];
dft1s=ones(dft);
add=[1394 6584 14428]; % # of additions in DFTs
mult=[198 792 1584]; % # of multiplications in DFTs
l=[3 4 5 6 7]'; % # of moduli (different cases)
l1s=ones(l);
big1s=l1s*dft1s;
bd=8; % # of bits in data
bc=8; % # of bits in coefficients
a=24; % # of trans in full adder
t=8;  % RC delay through full adder
b=(bd+bc)*dft1s+(log(dft)./log(2));
%
for i=1:(size(dft)*[0;1])
    d=dft(i);
    adds=add(i);
    mults=mult(i);
    for j=1:(size(l)*[1;0])
        ll=l(j);
        % find bb
        bb=b(i)/ll;
        %find at for dft and ll
        area=ll*(2*bb);
        time=(bb+1);
        rnsaddat(j,i)=area*time;
        area=ll*(4*bb+2*aromm((bb+1),bb));
        trnsm=2*bb+1+tromm((bb+1),bb);
        rnsmultat(j,i)=area*trnsm;
    end
end
%
intaddat=b.*b;
%
%rnsaddat=2b(b/l+1)*a*t;
```

```
%
intmultat=(b.*b-1.5*b).*(2*b+dft1s);
%+(6/24)*(b.*b).*(2*b+dft1s);
%
scadd=2*(l)*dft;
scmult=2*l*dft;
conadd=0*(l*dft);
%+3*2*(l-l1s)*dft;
conmult=0*(l*dft);
%
intat=intaddat.*add+intmultat.*mult;
for i=1:(size(l)*[1;0])
   for j=1:(size(dft)*[0;1])
   x(i,j)=aromm(bd,b(j))*tromm(bd,b(j))*dft(j)*2;
   end
end
y=rnsaddat.*((l1s*add)+scadd+conadd);
z=rnsmultat.*((l1s*mult)+scmult+conmult);
rnsat=x+y+z;
comp=[intat' rnsat'];
plot(dft,comp);
%title('Minimum Area-Time Products for WFTAs');
xlabel('WFTA Size');
ylabel('Minimum A*T (#FAs)');
labels=['Int';
        '3  ';
        '4  ';
        '5  ';
        '6  ';
        '7  '];
text(504*ones(comp(1,:)),comp(3,:)+[0 0 0 0 0 0], labels);
!del atcomp.met
meta atcomp
```

```
% atcomp2.m
% area-time comparison for DFT computation
% stricter control over moduli (integers)
dft=[63 252 504];
dft1s=ones(dft);
add=[1394 6584 14428]; % # of additions in DFTs
mult=[198 792 1584]; % # of multiplications in DFTs
l=[3 4 5 6 7]'; % # of moduli (different cases)
l1s=ones(l);
big1s=l1s*dft1s;
bd=8; % # of bits in data
bc=8; % # of bits in coefficients
a=24; % # of trans in full adder
t=8;  % RC delay through full adder
b=ceil((bd+bc)*dft1s+(log(dft)./log(2)));
%
for i=1:3
    d=dft(i);
    adds=add(i);
    mults=mult(i);
    for j=1:(size(l)*[1;0])
        l1=l(j);
        % find bb(k)
        bb(1)=ceil(b(i)/l1);
        bleft=b(i)-bb(1);
        for k=2:l(j)
            bb(k)=ceil(bleft/(l1-k+1));
            bleft=bleft-bb(k);
        end
        %find at for d and l1
        area=0;
        for k=1:l(j)
            area=area+2*bb(k);
        end
        time=(bb(1)+1);
        rnsaddat(j,i)=area*time;
        area=0;
```

```
        for k=1:l(j)
            szrnsm=4*bb(k)+2*aromm((bb(k)+1),bb(k));
            area=area+szrnsm;
        end
        trnsm=2*bb(1)+1+tromm((bb(1)+1),bb(1));
        rnsmultat(j,i)=area*trnsm;
    end
end
%
intaddat=b.*b;
%
%rnsaddat=2b(b/l+1)*a*t;
%
intmultat=(b.*b-1.5*b).*(2*b+dft1s);
%+(6/24)*(b.*b).*(2*b+dft1s);
%
scadd=2*(l)*dft;
scmult=2*l*dft;
conadd=0*(l*dft);
%+3*2*(l-l1s)*dft;
conmult=0*(l*dft);
%
intat=intaddat.*add+intmultat.*mult;
for i=1:(size(l)*[1;0])
   for j=1:(size(dft)*[0;1])
   x(i,j)=aromm(bd,b(j))*tromm(bd,b(j))*dft(j)*2;
   end
end
y=rnsaddat.*((l1s*add)+scadd+conadd);
z=rnsmultat.*((l1s*mult)+scmult+conmult);
rnsat=x+y+z;
comp=[intat' rnsat'];
plot(dft,comp);
%title('Minimum Area-Time Products for WFTAs');
xlabel('WFTA Size');
ylabel('Minimum A*T (#FAs)');
labels=['Int';
```

```
        '3  ';
        '4  ';
        '5  ';
        '6  ';
        '7  '];
text(504*ones(comp(1,:)),comp(3,:)+[0 0 0 0 0 0], labels);
!del atcomp2.met
meta atcomp2
```

```
% atcomp3.m
% area-time comparison for DFT computation
% more strict control of moduli ('fudge' loss in dyn range)
dft=[63 252 504];
% 1008];
dft1s=ones(dft);
add=[1394 6584 14428];
% 34416]; % # of additions in DFTs
mult=[198 792 1584];
% 3564]; % # of multiplications in DFTs
l=[3 4 5 6 7]'; % # of moduli (different cases)
l1s=ones(l);
big1s=l1s*dft1s;
bd=8; % # of bits in data
bc=8; % # of bits in coefficients
a=24; % # of trans in full adder
t=8;  % RC delay through full adder
b=(bd+bc)*dft1s+(log(dft)./log(2));
rnsb=ceil(l1s*b+(l.*[0 0 0.2 0.25 0.43]')*dft1s);
b=ceil(b);
%
for i=1:(size(dft)*[0 1]')
    d=dft(i);
    adds=add(i);
    mults=mult(i);
    for j=1:(size(l)*[1 0]')
        l1=l(j);
        % find bb(k)
        bb(1)=ceil(rnsb(j,i)/l1);

 %bb(1)=ceil(2*bb(1)-log(exp(bb(1)*log(2))-3)/log(2));
        bleft=rnsb(j,i)-bb(1);
        for k=2:l(j)
            bb(k)=(bleft/(l1-k+1));
            if bb(k)<2,
               bb(k)=2;
            else
```

```
                % z=log(exp(bb(k)*log(2))-3)/log(2);
                 bb(k)=ceil(bb(k));
            end
            bleft=bleft-bb(k);
        end
        %find at for d and ll
        area=0;
        for k=1:l(j)
            area=area+2*bb(k);
        end
        time=(bb(1)+1);
        rnsaddat(j,i)=area*time;
        area=0;
        for k=1:l(j)
            szrnsm=4*bb(k)+2*aromm((bb(k)+1),bb(k));
            area=area+szrnsm;
        end
        trnsm=2*bb(1)+1+tromm((bb(1)+1),bb(1));
        rnsmultat(j,i)=area*trnsm;
    end
end
%
intaddat=b.*b;
%
%rnsaddat=2b(b/l+1)*a*t;
%
intmultat=(b.*b-1.5*b).*(2*b+dft1s);
%+(6/24)*(b.*b).*(2*b+dft1s);
%
scadd=2*l*dft;
scmult=2*l*dft;
conadd=0*(l*dft);
%+3*2*(l-l1s)*dft;
conmult=0*(l*dft);
%
intat=intaddat.*add+intmultat.*mult;
for i=1:(size(l)*[1;0])
```

```
   for j=1:(size(dft)*[0;1])
   x(i,j)=aromm(bd,rnsb(i,j))*tromm(bd,rnsb(i,j))*dft(j)*2;
   end
end
y=rnsaddat.*((l1s*add)+scadd+conadd);
z=rnsmultat.*((l1s*mult)+scmult+conmult);
rnsat=x+y+z;
comp=[intat' rnsat'];
%
% Distributed
N=[64 256 512];
logn=[6 8 9];
a=aromm(4,(4*(bc+1)));
t=tromm(4,(4*(bc+1)));
size=(N/2).*(log(N)/log(2))*(a+(bc+1)*2);
time=t*(bd*ones(N)+(bc+1)*(log(N)/log(2)));
dist=size.*time;
%
plot(dft,comp,N,dist);
%title('Minimum Area-Time Products for WFTAs');
xlabel('WFTA Size');
ylabel('Minimum A*T (#FAs)');
labels=['Int';
        '3  ';
        '4  ';
        '5  ';
        '6  ';
        '7  '];
text(504*ones(comp(1,:)),comp(3,:)+[0 0 0 0 0 0], labels);
!del atcomp3.met
meta atcomp3
```

```
% atcomp4.m
% area-time comparison for DFT computation
% more strict control of moduli (fewer integers)
% smaller range for integers
dft=[63 252 504];
% 1008];
dft1s=ones(dft);
add=[1394 6584 14428];
% 34416]; % # of additions in DFTs
mult=[198 792 1584];
% 3564]; % # of multiplications in DFTs
l=[3 4 5 6 7]'; % # of moduli (different cases)
l1s=ones(l);
big1s=l1s*dft1s;
bd=8; % # of bits in data
bc=8; % # of bits in coefficients
a=24; % # of trans in full adder
t=8;  % RC delay through full adder
b=(bd+bc)*dft1s+(log(dft)./log(2));
rnsb=ceil(l1s*b+(1.*[0 0 0.2 0.25 0.43]')*dft1s);
%
for i=1:(size(dft)*[0 1]')
    d=dft(i);
    adds=add(i);
    mults=mult(i);
    for j=1:(size(l)*[1 0]')
        ll=l(j);
        % find bb(k)
        bb(1)=ceil(rnsb(j,i)/ll);

 %bb(1)=ceil(2*bb(1)-log(exp(bb(1)*log(2))-3)/log(2));
        bleft=rnsb(j,i)-bb(1);
        for k=2:l(j)
            bb(k)=(bleft/(ll-k+1));
            if bb(k)<2,
               bb(k)=2;
            else
```

```
                % z=log(exp(bb(k)*log(2))-3)/log(2);
                 bb(k)=ceil(bb(k));
            end
            bleft=bleft-bb(k);
        end
        %find at for d and ll
        area=0;
        for k=1:l(j)
            area=area+2*bb(k);
        end
        time=(bb(1)+1);
        rnsaddat(j,i)=area*time;
        area=0;
        for k=1:l(j)
            szrnsm=4*bb(k)+2*aromm((bb(k)+1),bb(k));
            area=area+szrnsm;
        end
        trnsm=2*bb(1)+1+tromm((bb(1)+1),bb(1));
        rnsmultat(j,i)=area*trnsm;
    end
end
b=ceil(bd*dft1s+max((log(dft)/log(2)),(bc*dft1s)));
%
intaddat=b.*b;
%
%rnsaddat=2b(b/l+1)*a*t;
%
intmultat=(b.*b-1.5*b).*(2*b+dft1s);
%+(6/24)*(b.*b).*(2*b+dft1s);
%
scadd=2*(l)*dft;
scmult=2*l*dft;
conadd=0*(l*dft);
%+3*2*(l-l1s)*dft;
conmult=0*(l*dft);
%
intat=intaddat.*add+intmultat.*mult;
```

```
for i=1:(size(l)*[1;0])
   for j=1:(size(dft)*[0;1])
   x(i,j)=aromm(bd,rnsb(i,j))*tromm(bd,rnsb(i,j))*dft(j)*2;
   end
end
y=rnsaddat.*((l1s*add)+scadd+conadd);
z=rnsmultat.*((l1s*mult)+scmult+conmult);
rnsat=x+y+z;
comp=[intat' rnsat'];
plot(dft,comp);
%title('Minimum Area-Time Products for WFTAs');
xlabel('WFTA Size');
ylabel('Minimum A*T (#FAs)');
labels=['Int';
        '3  ';
        '4  ';
        '5  ';
        '6  ';
        '7  '];
text(504*ones(comp(1,:)),comp(3,:)+[0 0 0 0 0 0], labels);
!del atcomp4.met
meta atcomp4
```

Bibliography

[1] Kevin S. Anderson. Real time considerations for DFT algorithms. *Proceedings of the SPIE— The International Society for Optical Engineering*, 431:230–238, 1983.

[2] John D. Blanken. Efficient computer implemenations of fast fourier transforms. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB, Ohio, December 1980.

[3] C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms.* John Wiley and Sons, New York, 1985.

[4] David A. Hodges and Horace G. Jackson. *Analysis and Design of Digital Integrated Circuits.* McGraw-Hill Book Company, New York, second edition, 1988.

[5] C. H. Huang, A. H. Gelders, and D. G. Peterson. Residue processing. Technical report, Lockheed Missiles and Space Company, Inc., January 1985.

[6] Chao H. Huang and Fred J. Taylor. New technique for WFTA input/output reordering. *International Journal of Computer and Information Sciences*, 10(1):27–37, February 1981.

[7] W. K. Jenkins. The use of residue number systems in the design of FIR digital filters. *IEEE Transactions on Circuits and Systems*, CAS-24:191–201, April 1977.

[8] G. A. Jullien. Implementation of multiplication, modulo a prime number, with applications to number theoretic transforms. *IEEE Transactions on Computers*, C-29:899–905, October 1980.

[9] M. D. MacLeod and N. L. Bragg. Fast hardware implementation of the Winograd Fourier Transform Algorithm. *Electronics Letters*, 19(10):363–365, May 1983.

[10] Alan V. Oppenheim and Ronald W. Schaefer. *Discrete-Time Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[11] M. A. Soderstrand. A new hardware implementation of modular adders for residue number systems. In *Proceedings of the 26th Midwest Symposium on Circuits and Systems*, pages 412–415. Western Periodicals, 1983.

[12] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing.* IEEE Press, New York, 1986.

[13] M. A. Soderstrand and Carmel Vernia. A high-speed low-cost modulo p_i multiplier with rns arithmetic applications. *Proceedings of IEEE*, 68:529–532, April 1980.

[14] N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and Its Applications to Computer Technology.* McGraw-Hill, New York, 1967.

[15] F. J. Taylor. Large moduli multipliers for signal processing. *IEEE Transactions on Circuits and Systems*, CAS-28:731–736, July 1981.

[16] F. J. Taylor. Residue arithmetic: A tutorial with examples. *IEEE Transactions on Computers*, 17(5):50–62, May 1984.

[17] Fred J. Taylor. A comparison of DFT algorithms using a residue architecture. *International Journal on Computers and Electrical Engineering*, 8(3):161–173, September 1981.

[18] Kou-Hu Tzou and Nigel P. Morgan. Fast pipelined DFT processor and its programming considerations. *IEE Proceedings, Part G*, 132(6):273–276, December 1985.

[19] John C. Vaccaro, Bruce L. Johnson, and Carol L. Nowacki. A systolic discrete fourier transform using residue number systems over the ring of Gaussian integers. In *Proceedings— ICASSP, IEEE International Conference on Acoustics Speech and Signal Processing*, volume 2, pages 1157–1160, Tokyo, April 1986.

[20] J. S. Ward, J. V. McCanny, and J. G. McWhirter. Bit-level systolic array implementation of the Winograd Fourier Transform Algorithm. *IEE Proceedings, Part F: Communications, Radar, and Signal Processing*, 132(6):473–479, October 1985.

[21] Neil H. E. Weste and Kamram Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.